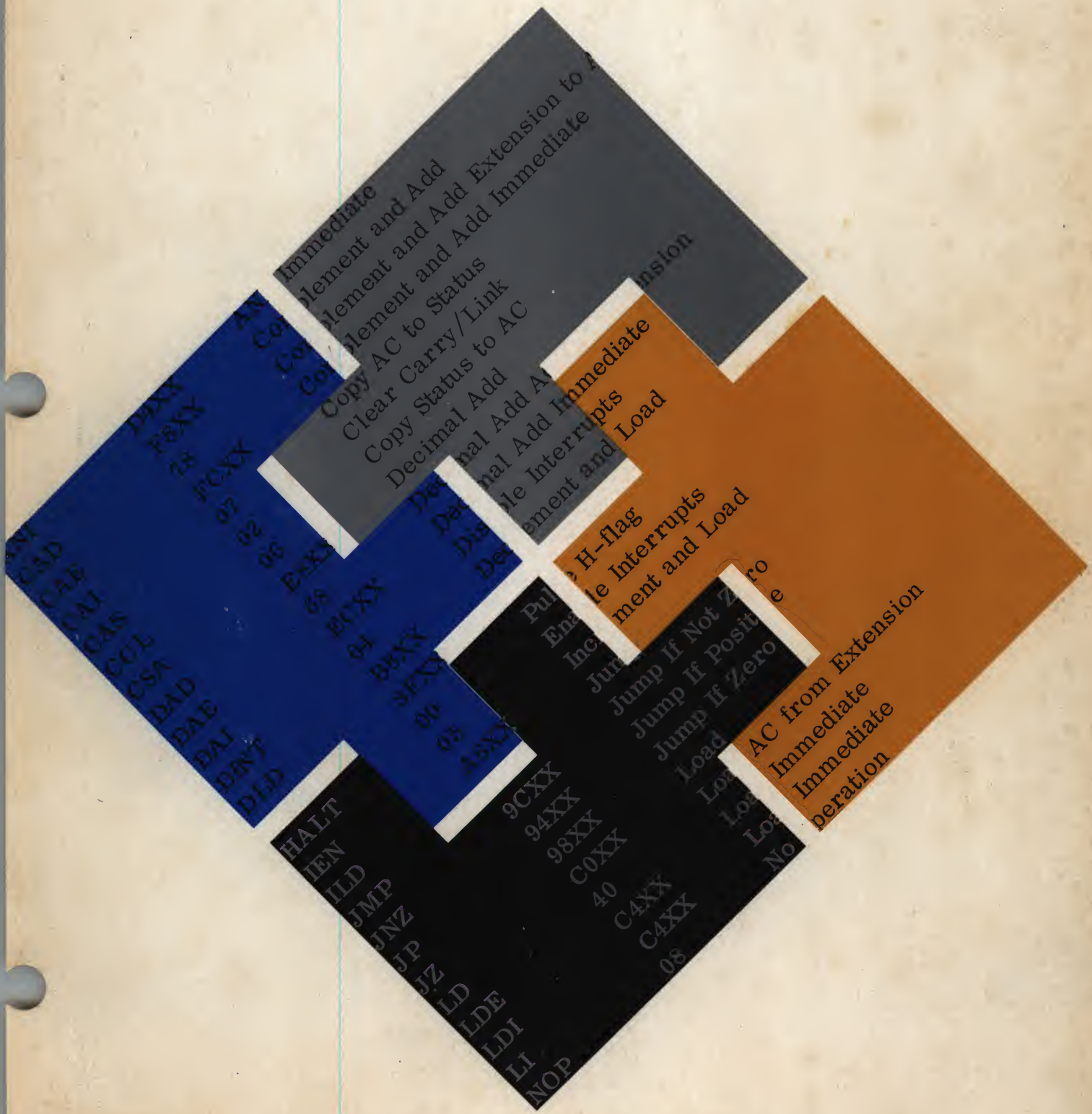
SC/MP

Microprocessor
Assembly Language
Programming
Manual

National
Semiconductor



Publication Number 4200094C
Order Number ISP-8S/994Y
January 1977

SC/MP
Simple Cost-effective
MicroProcessor

Assembly Language
Programming Manual

2900 Semiconductor Drive
Santa Clara, California 95051

PREFACE

The SC/MP Assembly Language Programming Manual provides tutorial and reference information required for writing user application programs. Component data sheets and information pertaining to prototyping systems are available in other documents.

It is suggested that the reader thoroughly review the table of contents, illustrations, and tables to familiarize himself with an overview of the organization of this manual before reading its contents. By so doing, the reader will have an understanding of the extent-of-coverage.

This document supersedes the SC/MP Programming and Assembler Manual (Publication Number 4200094B).

The material in this manual is for information purposes only and is subject to change without notice.

Changes to this document will be reported in COMPUTE, the Microprocessor Users Group newsletter.

Copies of this publication and other National Semiconductor publications may be obtained from the National Semiconductor sales office or distributor serving your locality.

Reference publications:

- SC/MP (FORTRAN) Cross Assembler Software Package, Installation and Operating Instructions (Publication Number 4200125)
- SC/MP (IMP-16) Cross Assembler Operating Instructions (Publication Number 4200126)
- SC/MP (PACE) Cross Assembler Operating Instructions (Publication Number 420305238-001)

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

Chapter 1

GENERAL INFORMATION

SC/MP is a Simple Cost-effective MicroProcessor that affords the user with a powerful, versatile, and easy-to-use repertoire of instructions. These instructions are represented as simple mnemonics in the SC/MP assembly language. Writing user computer programs in this assembly language increases programming efficiency considerably.

Other publications that pertain to the SC/MP Assembler are listed in the preface.

The following is a brief description of the contents of chapter 2 through 7.

Chapter 2, SC/MP Microprocessor Overview, describes the registers available to the user, the types of addressing used in SC/MP, and SC/MP input/output and interrupt facilities.

Chapter 3, Assembly Language, describes the elements, the structure, and the coding conventions of the SC/MP Assembly Language.

Chapter 4, Statements, is a detailed description of the characteristics and operation of the SC/MP instruction set. The chapter also describes four types of assembler-dependent statements: the comment, the pseudo-instruction, the assignment, and the directive.

Chapter 5, Programming Techniques, provides programming examples that show how to write efficient code, how tc link to subroutines, and how to perform input/output (programmed and interrupt).

Chapter 6, Macros, defines macros, what they are, how to use them, and how to write them.

Chapter 7, Assembler Input/Output Formats, defines the input/output formats that are common to all SC/MP assemblers.

Chapter 2

SC/MP MICROPROCESSOR OVERVIEW

This chapter describes the main features of the SC/MP microprocessor. Only those features with which the programmer is primarily concerned are discussed. Detailed information on the SC/MP device is contained in the SC/MP Data Sheet.

2.1 OPERATIONAL FEATURES

The SC/MP is a programmable 8-bit parallel processor with 16-bit memory and peripheral addressing. Functionally, SC/MP has a bidirectional data bus connecting the CPU, memory, and peripheral devices. Peripheral devices are assigned memory addresses, and any standard memory-reference instruction can be used for input/output operations. Memory is expandable to 65,536 bytes. Table 2-1 lists the operational features of SC/MP.

Table 2-1. Operational Features

Data Length	8 Bits (Byte)
Instruction Set	46 Instructions
Arithmetic	Parallel, binary, fixed point, twos complement 2-digit BCD addition
Memory	Up to 65,536 bytes of ROM/RAM
Registers	One 8-bit Accumulator One 8-bit Status Register One 8-bit Extension Register Four 16-bit Pointer Registers (one is the Program Counter)
Addressing Modes	Program Counter Relative Indexed Auto-indexed Immediate
Input/Output and Control	16-bit Address Bus: 4 multiplexed bits and 12 static bits 8-bit Bidirectional Data Bus

2.2 REGISTERS

The seven registers available to the programmer who uses the SC/MP assembly language are shown in figure 2-1 and are discussed in the following paragraphs.

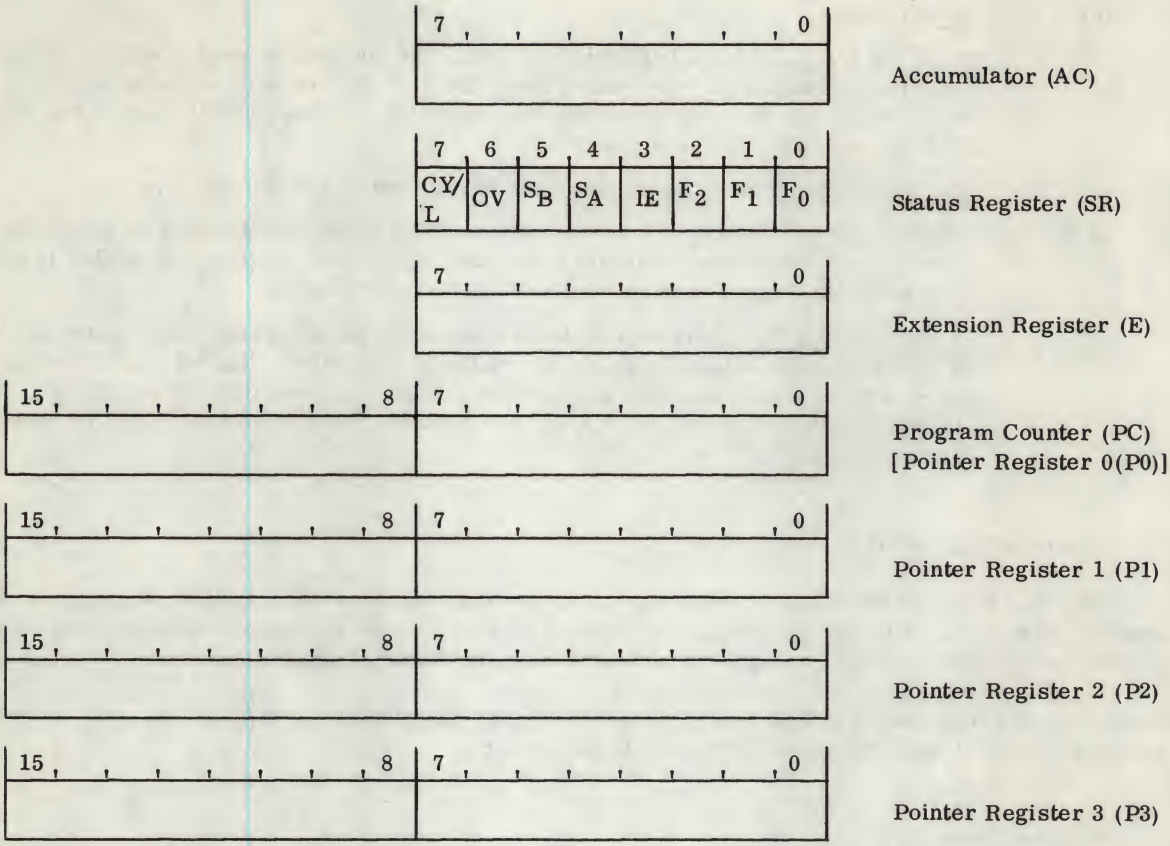


Figure 2-1. SC/MP Registers

2.2.1 Accumulator (AC)

The 8-bit Accumulator (AC) is the primary working register of SC/MP. The accumulator is used in performing arithmetic and logic operations and for storing the results of those operations. Data transfers, shifts, and rotates also use the accumulator. In all, 37 of the 46 SC/MP instructions use the accumulator.

2.2.2 Status Register (SR)

The Status Register (SR) provides storage for arithmetic, control, and software status flags. The bit position and function of each flag in the register are shown and defined below.

7	6	5	4	3	2	1	0	Bit Positions
CY/L	OV	S_B	S_A	IE	F_2	F_1	F_0	Flags

Bit	Description
0	User Flag 0 (F0). User assigned for control function or for software status. The output of this bit is available at a pin of the SC/MP device.
1	User Flag 1 (F1). Same as F0.
2	User Flag 2 (F2). Same as F0.
3	Interrupt Enable Flag (IE). The processor recognizes the interrupt input if this flag is set.

Bit	Description
4	Sense Bit A (SA). This bit is tied to a package pin and may be used to sense external conditions. This bit is "read-only"; thus, the Copy Accumulator to Status Register Instruction (CAS) does not affect this bit. When the Interrupt Enable Flag is set, Sense Bit A serves as the interrupt input.
5	Sense Bit B (SB). Same as SA, except it is not used as an interrupt input.
6	Overflow (OV). This bit is set if an arithmetic overflow occurs during an add (ADD, ADI, or ADE) or a complement-and-add instruction (CAD, CAI, or CAE). Overflow is not affected by the decimal-add instructions (DAD, DAI, or DAE).
7	Carry/Link (CY/L). This bit is set if a carry from the most significant bit occurs during an add, a complement-and-add, or decimal-add instruction. The bit is also included in the Shift Right with Link (SRL) and the Rotate Right with Link (RRL) Instructions. CY/L is input as a carry into the bit 0 position of the add, complement-and-add, and decimal-add instructions.

2.2.3 Extension Register (E)

The 8-bit Extension Register (E) is used primarily with the accumulator to perform arithmetic, logic, and data-transfer operations. If the displacement in an indexed or an auto-indexed memory-reference instruction equals -128_{10} (80_{16}), then the contents of E are substituted for the displacement for the given instruction.

Another function of the Extension Register is serial input/output. This feature is explained in detail in the description of the Serial Input/Output Instruction (SIO) in chapter 4.

2.2.4 Program Counter (PC)

The Program Counter (PC) is the dedicated 16-bit Pointer Register P0. The Program Counter contains the address of the instruction being executed. In the event of an interrupt or a subroutine call, the contents of the Program Counter may be stored on a software stack and may be retrieved subsequently when the processor returns to the main program. The use of a software stack is explained in chapter 5.

The Program Counter is incremented just before an instruction fetch. Therefore, the effective address of any transfer-of-control should be one less than the actual address to be executed (taking into account the modulo 2^{12} address arithmetic as explained in section 2.3).

Arithmetic affecting the Program Counter is performed on the low-order 12 bits; the high-order 4 bits are not affected. A further explanation of this may be found in section 2.3.

2.2.5 Pointer Registers (P1, P2, P3)

There are three 16-bit Pointer Registers available for addressing memory and peripherals, and for use as page pointers, stack pointers, or index registers. In simple applications the Pointer Registers might be used as temporary storage registers. Typically, the programmer assigns a specific function to each register. The following assignments might be used.

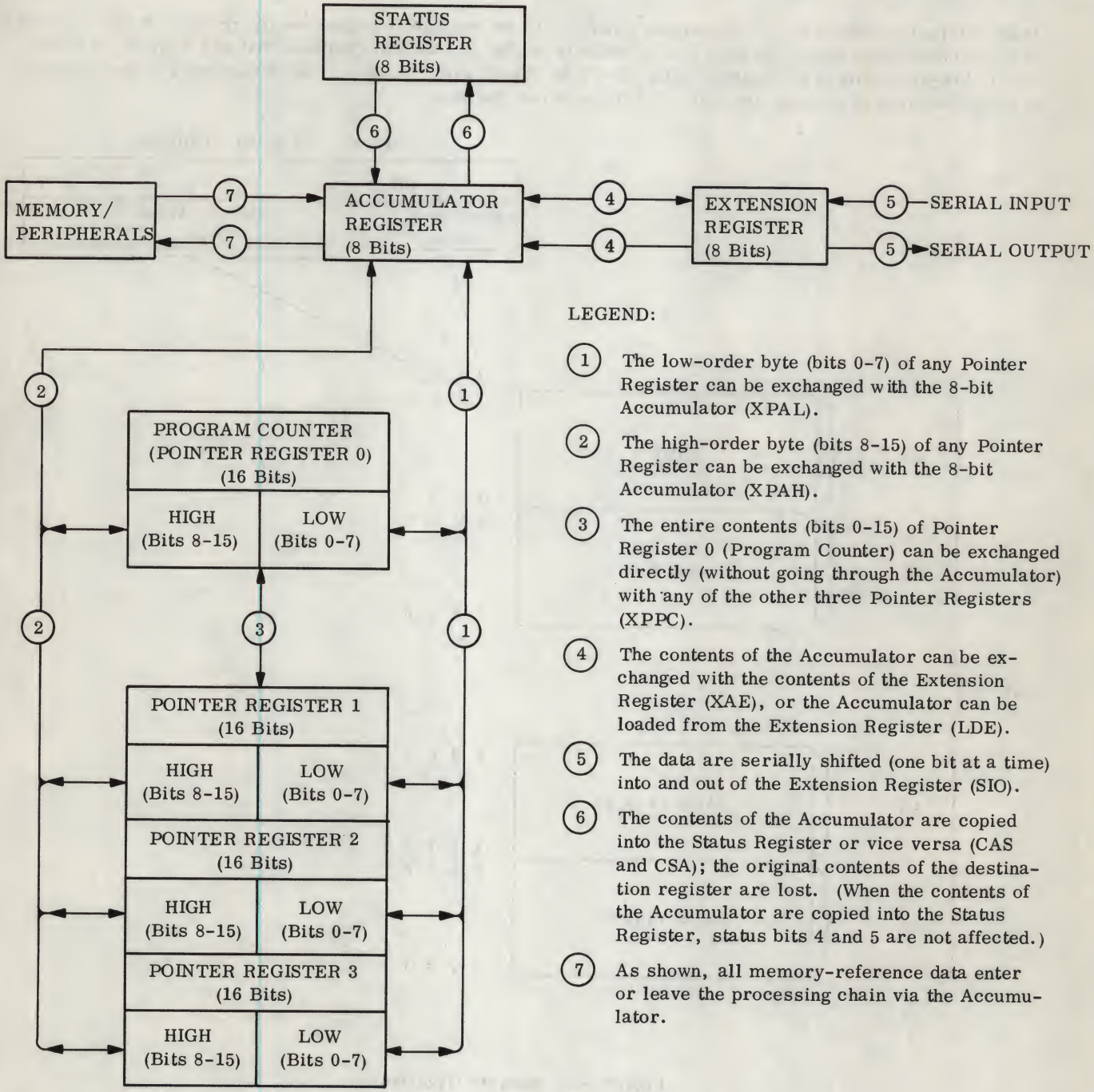
P1 — ROM Pointer
P2 — Stack Pointer
P3 — Subroutine Pointer

NOTE

The SC/MP hardware uses Pointer Register 3 when servicing an interrupt. For details, see 2.5.

2.2.6 Inter-Register Data Flow

Data flow relationships between memory and the seven registers of SC/MP are shown and described in figure 2-2.



LEGEND:

(1) The low-order byte (bits 0-7) of any Pointer Register can be exchanged with the 8-bit Accumulator (XPAL).

(2) The high-order byte (bits 8-15) of any Pointer Register can be exchanged with the 8-bit Accumulator (XPAH).

(3) The entire contents (bits 0-15) of Pointer Register 0 (Program Counter) can be exchanged directly (without going through the Accumulator) with any of the other three Pointer Registers (XPPC).

(4) The contents of the Accumulator can be exchanged with the contents of the Extension Register (XAE), or the Accumulator can be loaded from the Extension Register (LDE).

(5) The data are serially shifted (one bit at a time) into and out of the Extension Register (SIO).

(6) The contents of the Accumulator are copied into the Status Register or vice versa (CAS and CSA); the original contents of the destination register are lost. (When the contents of the Accumulator are copied into the Status Register, status bits 4 and 5 are not affected.)

(7) As shown, all memory-reference data enter or leave the processing chain via the Accumulator.

Figure 2-2. Interrelationship of SC/MP Registers

2.3 MEMORY ORGANIZATION AND ADDRESSING

Memory is organized as a sequence of 8-bit bytes. Each byte is identified by a 16-bit address that represents its sequential position in memory from 0 to $FFFF_{16}$ ($65,535_{10}$).

In the internal architecture of the computer, memory is divided into 16 pages (see figure 2-3). A page is a set of 4,096 consecutive bytes, the first byte of which is located at a memory address that is a multiple of 4,096. The 12 low-order bits of the address of the first byte of each page are zeros. Each memory address consists of a 4-bit address of the page and a 12-bit address within the page.

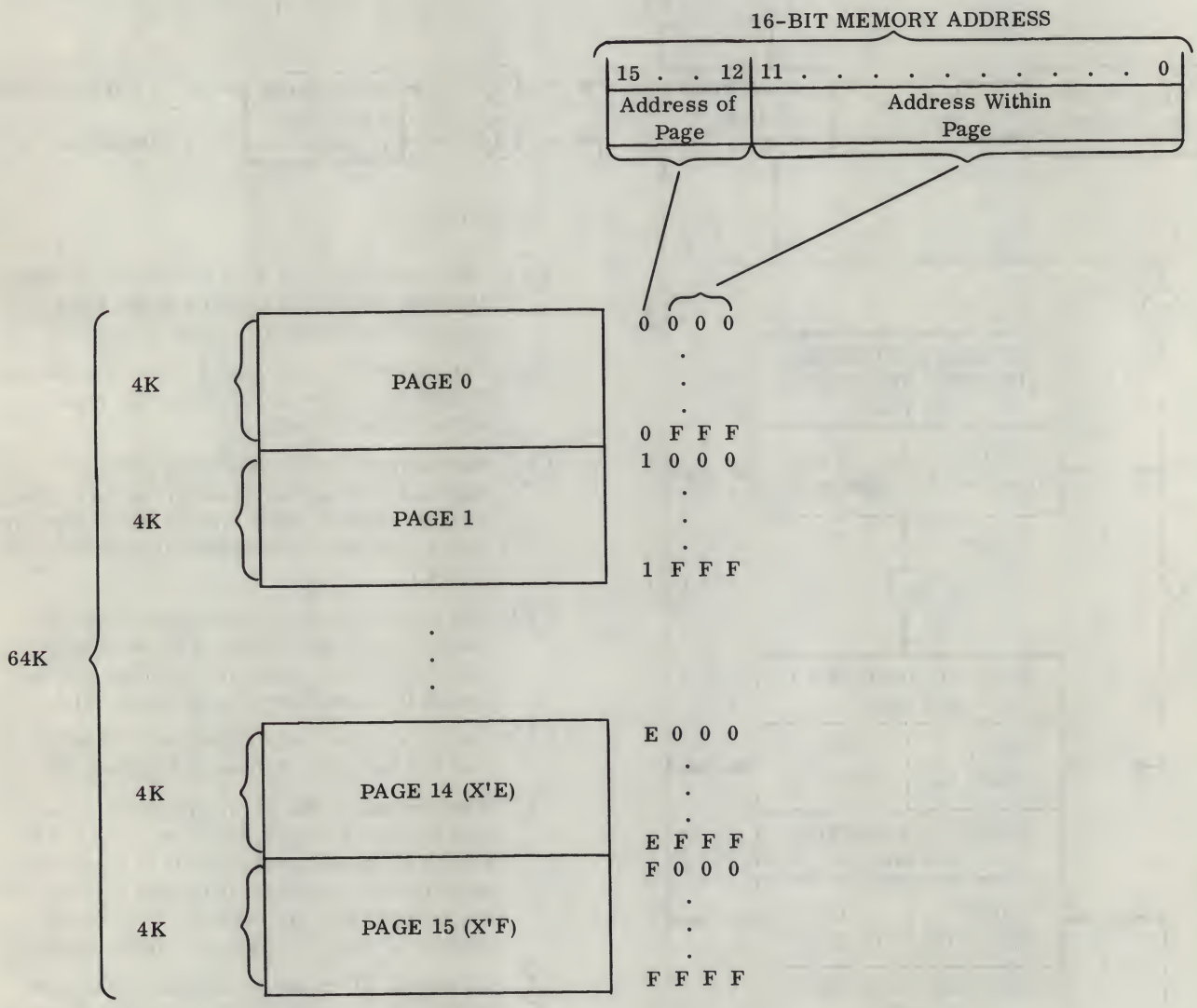


Figure 2-3. Memory Organization

2.3.1 Paging Considerations

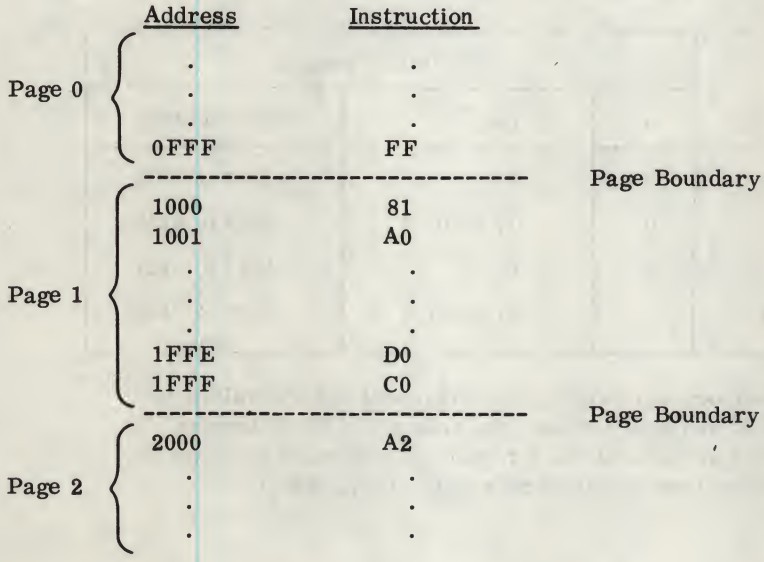
When performing arithmetic to calculate the effective address of an operand, the calculations are performed on the low-order (displacement) portion of the address with no carry into the high-order (page) portion. For example:

	Current Address		Displacement from Current Address	Effective Address	
	Address of Page	Address Within Page		Address of Page	Address Within Page
Calculated Effective Address Remains Within Page	1	FB4	05	1	FB9
Calculated Effective Address Crosses Page Boundary	1	FB4	4D	1	001 (wrap around)

In the case when the calculated effective address crosses the page boundary of the current page, a "wrap around" occurs.

When incrementing the address to fetch the next instruction, the same page/displacement arithmetic occurs.

If a 2-byte instruction is inadvertently separated by a page boundary, an error occurs. Consider the following sequence of instructions on pages 0, 1, and 2 — with the first digit of the address designating the page and the next three digits, the location within the page.



The instruction intended, when the PC = 1FFF (last word in page 1), is X'C0A2 (LD X'20A2). However, instead of fetching the latter half of the instruction from page 2, a wrap-around is made to the first word of page 1; the instruction that will be executed is X'C081 (LD X'1081). The SC/MP assembler assumes the user will organize his programs in pages of 4,096 bytes for each page to provide protection from the situation described above. If a boundary cross-over occurs, the assembler issues an alarm message.

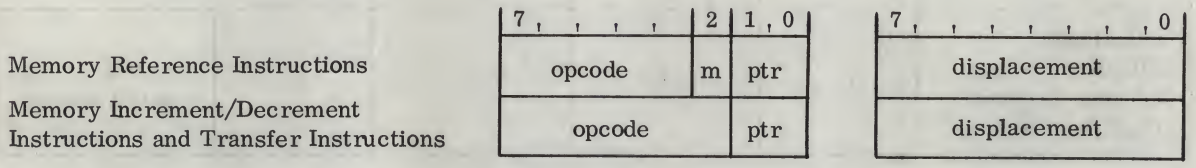
If a user wants to start or continue a program from the beginning of a new page, he must explicitly set the location counter (see 4.4, Assignment Statement) and he must use the Jump to Subroutine (JS) pseudo instruction (see 4.3, Pseudo Instruction). For example:

```
        JS      P3,PAGE3            ;JUMP TO PAGE 3
        :
        :
PAGE3:  .=X'2000
```

This example jumps to the first location of page 3. The user should be aware that the contents of the Accumulator are lost.

2.3.2 Methods of Addressing

During execution, instructions and data defined in a program are stored into and loaded from specific memory locations, the accumulator, or selected registers. Because the CPU, memory (read/write and read-only), and peripherals are on a common data bus, any instruction used to address memory may also be used to address the peripherals. The formats of the instruction groups that reference memory are shown below.



Memory-reference instructions use the PC-relative, indexed, or auto-indexed methods of addressing memory. The memory increment/decrement instructions and the transfer instructions use the PC-relative or indexed methods of addressing.

The various methods of addressing memory and peripherals are shown in table 2-2.

Immediate addressing is an addressing mode specific to the immediate instruction group.

Table 2-2. Addressing Modes

Type of Addressing	Operand Formats		
	m	ptr	displacement
PC-relative	0	0	-128* to +127
Indexed	0	1, 2, or 3	-128* to +127
Immediate	1	0	-128* to +127
Auto-indexed	1	1, 2, or 3	-128* to +127

*Note: For memory-reference instructions that are PC-relative, indexed, and auto-indexed, the contents of the Extension Register are substituted for the displacement if the value of the instruction displacement equals -128_{10} (80_{16}).

2.3.2.1 PC-Relative Addressing

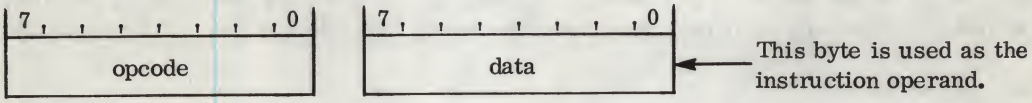
A PC-relative address is formed by adding the displacement value specified in the operand field of the instruction to the current contents of the program counter. The displacement is an 8-bit twos-complement number, so the range of the PC-relative addressing format is -128_{10} to $+127_{10}$ bytes from the current location of the Program Counter. During execution of an instruction, the program counter contains the address of the last byte of the instruction. In the assembly language source statement, a symbolic expression representing an address normally is used in the operand field. In this case, the assembler generates the displacement from the current address to the address specified by the symbolic expression. The following examples show the use of PC-relative addressing.

```
Location    Generated
Counter     Code
0005        C00E        LOOP:   LD      TEMP    ;LOAD THE VALUE IN TEMPORARY STORAGE
                                .
                                .
                                .
000E        90F5                JMP     LOOP    ;REPEAT
                                .
                                .
                                .
0014        04          TEMP:   .BYTE   X'04
```

The assembler assumes PC-relative addressing in the memory-reference and transfer instructions when no pointer-register operand is specified.

2.3.2.2 Immediate Addressing

Immediate addressing uses the value in the second byte of a double-byte instruction as the operand for the operation to be performed (see below).



byte 1 byte 2

For example, compare a Load (LD) Instruction to a Load Immediate (LDI) Instruction. The Load Instruction uses the contents of the second byte of the instruction in computing the effective address of the data to be loaded. The Load Immediate Instruction uses the contents of the second byte as the data to be loaded. Because the operand occurs as the second byte of a two-byte instruction, page boundary conditions should be observed as mentioned in 2.3.1.

2.3.2.3 Indexed Addressing

Indexed addressing enables the programmer to address any location in memory through the use of the pointer register and the displacement. When indexed addressing is specified in an instruction, the contents of the designated pointer register are added to the displacement to form the effective address. The contents of the pointer register are not modified by indexed addressing. Indexed addressing is used to access tables or subroutines, to transfer control to another page, or to transfer control to a section of the current page that is outside the range of the PC-relative transfer. The rules for page boundaries still apply, so the user is cautioned about crossing page boundaries when using indexed addressing to access tables. Such a reference results in a wrap-around from the end to the beginning of the page, or vice-versa (2.3.1).

2.3.2.4 Auto-Indexed Addressing

Auto-indexed addressing provides two capabilities: (1) the ability to reference (load, store, and so forth) a memory location specified by the contents of a designated pointer register, and (2) the ability to increment or decrement the contents of the designated pointer register by the value of the displacement.

If the displacement is less than zero, the pointer register is decremented by the displacement before the contents of the effective address are fetched or stored. If the displacement is equal to or greater than zero, the pointer register is used as the effective address, and the pointer register is incremented by the displacement after the contents of the effective address are fetched or stored.

NOTE

The contents of the pointer register are modified by auto-indexed addressing.

An at sign (@) before the displacement operand designates an auto-indexed operation. Example:

```
Generated
Code
C601        LD      @1(P2)          ;GET A BYTE FROM THE TABLE, AUTO-INDEX
```

2.4 INPUT/OUTPUT FACILITIES

SC/MP uses a single bidirectional input/output bus to interconnect the CPU, memory, and peripheral devices. Peripheral devices are assigned memory addresses, so standard memory-reference instructions can be used for input/output operations.

Peripheral device addressing, data exchange, status reporting, and control-signal operations are performed by an external device controller. Because of variations in peripheral devices, depending on the function performed, a standard input/output operation cannot be described here. Similarly, the device controller operations vary widely, depending on the peripheral device being serviced. SC/MP provides the following facilities, which may be used in various applications, providing they match the device controller in use. Refer to figure 2-4.

- 16-bit Address
- 8-bit Parallel Input/Output
- 1-bit Serial Input
- 1-bit Serial Output
- 3 Flag Outputs
- 2 Sense Inputs

2.4.1 Address Lines

The 12-bit address lines contain the displacement portion of the effective memory address generated in response to a memory-reference instruction. The 4-bit page portion of the effective memory address is output on the data bus (see 2.4.4). Because peripheral devices are assigned memory addresses, a decoder in the associated device controller looks for its address or addresses. (Multiple addresses may be assigned to multifunction devices.) Details of interfacing hardware to SC/MP may be found in the SC/MP Data Sheet.

2.4.2 Parallel Input/Output

An 8-bit bidirectional data bus transfers data between the peripheral device controller and the processor. These data are associated with the memory-reference instruction that addressed the peripheral device. It is the function of the peripheral-device controller to place data on the line when the device is addressed for input and to transfer data from the lines when the device is addressed for output. The direction of the data transfer depends on the nature of the memory-reference instruction.

2.4.3 Serial Input/Output

Serial input/output is provided by using one of the eight data lines, or dedicated flag and sense input, or the Extension Register as a serial input/output shift register.

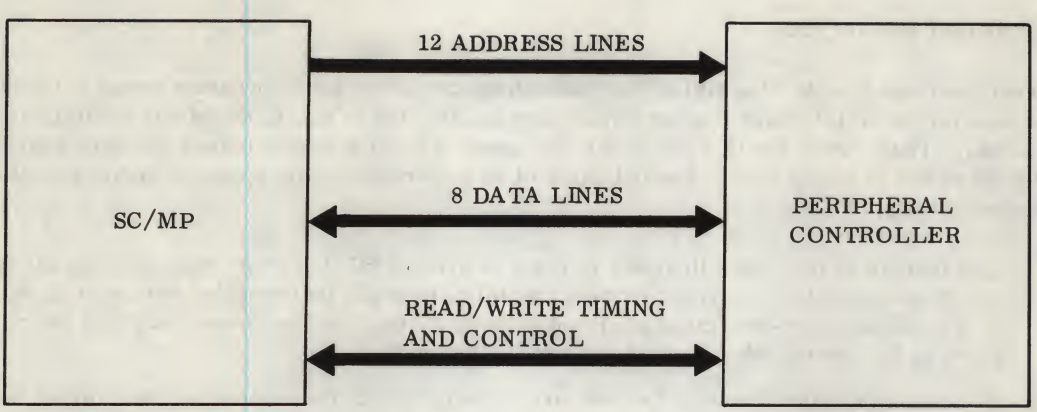


Figure 2-4. Interface to Peripheral Device Controller

In systems that have only one serial input/output device, the serial input and output pins may be tied directly to the input/output device and no address decoding is necessary. The Serial Input/Output Instruction (SIO) then is used for serial input/output. Timing may be provided by program loops using the delay instruction or by an external timing element that is tested by the jump-condition inputs. For asynchronous systems, a flag may be pulsed each time a new bit is shifted in/out, and a sense condition tested to detect bit received/ready.

Systems that have several serial input/output devices must be multiplexed, and device selection may be provided by the control flags, or by use of the parallel input/output commands to load an external latch.

The serial-data input and output pins may be used as sense-input and flag-output lines in systems that do not require the serial input/output function.

2.4.4 Input/Output Status

The input/output status (figure 2-5) is output on the data bus, along with the appropriate timing information on the timing lines, so the peripheral controller has the additional information available if it is required by a particular system. Two of the input/output status bits are for hardware functions (R-flag and I-flag); the remaining 6 input/output status bits are generated under software control.

7							0
H	D	I	R	A15	A14	A13	A12

A15 to A12 — Four most significant (page) address bits
H — H-flag generated by HALT instruction
D — D-flag generated by DLY instruction
I — I-flag generated by hardware for instruction fetch cycle
R — R-flag generated by hardware for read input/output cycle

Figure 2-5. Input/Output Status

2.5 INTERRUPT FACILITIES

When the Internal Interrupt Enable Flag (IE) in the Status Register is set under program control, the Sense A line is enabled to serve as an interrupt request input; when the IE Flag is reset, SC/MP is inhibited from detecting interrupts. Thus, while the IE Flag is set, the Sense A input is tested before the fetch phase of each instruction as shown in figure 2-6. Upon detection of an interrupt request (Sense A high), the following events occur automatically.

1. The IE Flag of the Status Register is reset to prevent SC/MP from responding to any further interrupt requests. Interrupt request capability then can be reenabled during or at the end of the ensuing user-generated interrupt service routine via the Enable Interrupt Instruction (IEN) or by copying the Accumulator into the Status Register.
2. The contents of the Program Counter are exchanged with the contents of the Pointer Register 3.
3. The contents of the Program Counter are incremented by one to address the first instruction of the user-generated interrupt service routine.

In order to enable interrupts, the interrupt system must be armed. This is accomplished, first, by setting the interrupt enable bit in the Status Register to '1' (by executing an Enable Interrupt Instruction or a Copy Accumulator to Status Register Instruction) and, then, by fetching and executing one additional instruction. A return from interrupt, therefore, can be accomplished by the Enable Interrupt Instruction (IEN) followed by Exchange Pointer 3 and Program Counter Instruction (XPPC 3). Interrupts are disarmed by setting the interrupt enable bit in the Status Register to '0' (by executing a Disable Interrupt Instruction (DINT) or a Copy Accumulator to Status Register Instruction (CAS)).

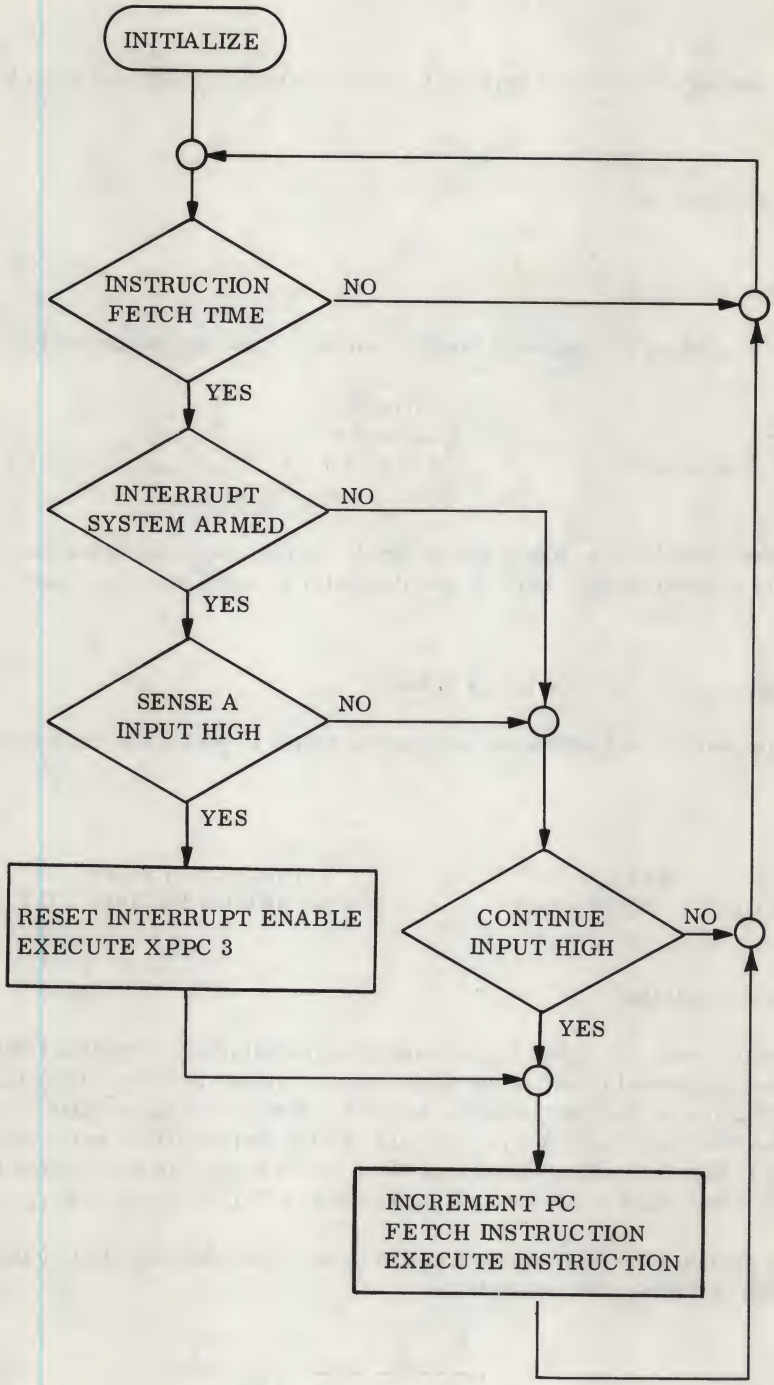


Figure 2-6. SC/MP Interrupt/Instruction-Fetch Process

Chapter 3

ASSEMBLY LANGUAGE

SC/MP assembly language statements have well-defined formats constructed from the elements described in this chapter.

3.1 BASIC ELEMENTS

3.1.1 Character Set

Statements are written using the following letters, numbers, and special characters:

Letters:	A through Z
Numbers:	0 through 9
Special Characters:	! $ % & ' ( ) * + , - . / : ; < = > @ b # ∧
	Note: b means blank

Except for the lower-case letters, any of the printable characters listed in appendix A, table A-1, ASCII Character Set in Hexadecimal Representation, may be specified with a .ASCII directive (see 4.5.9).

Example:

```
MSG3: .ASCII   'ORDER # 326-001'
```

Nonprintable (or printable) characters may be specified with a .BYTE directive (see 4.5.5) or a .DBYTE directive (see 4.5.6).

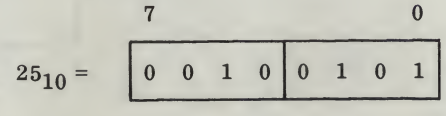
Examples:

```
CR:    .BYTE   0D          ;CARRIAGE RETURN
CRLF:  .DBYTE  0D0A        ;CARRIAGE RETURN/LINE FEED
```

3.1.2 Data Representation

Data are represented internally in SC/MP in twos-complement integer notation and binary-coded-decimal (BCD) notation. In twos-complement notation, the negative of a number is formed by complementing each bit in the data word and adding one to the complemented number. The sign of the number is indicated by the most significant bit. When the most significant bit is a '0', the number is positive or zero; when the most significant bit is a '1', the number is negative. Maximum range for a 16-bit number in this system is $7FFF_{16}$ ($+32767_{10}$) to 8000_{16} (-32768_{10}). Maximum range for an 8-bit number is $7F_{16}$ ($+127_{10}$) to 80_{16} (-128_{10}).

BCD notation is a system of number representation in which the decimal digits 0 through 9 are represented by a group of four binary digits. For example:



The maximum range of BCD values is 0 to 99_{10}. To obtain a BCD constant, the user may code one or two digits (0 through 9) as a hexadecimal constant, for example, 059.

3.1.3 Terms

The relationship of terms is shown in figure 3-1. The various types of terms are described in the following paragraphs.

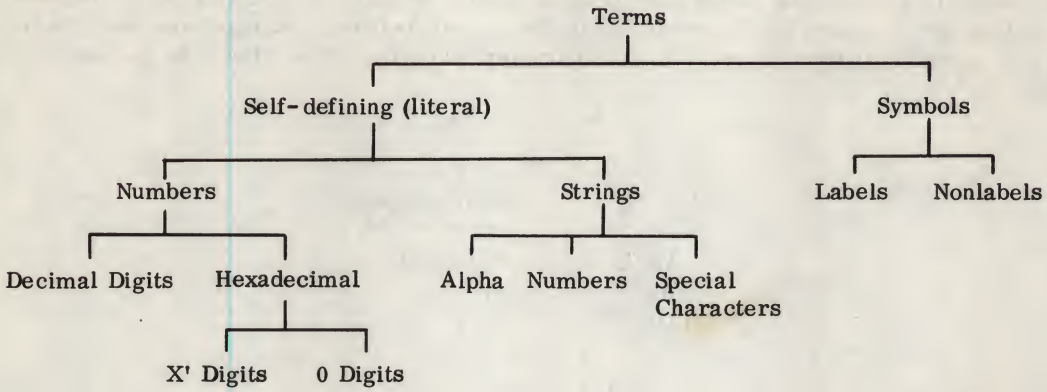


Figure 3-1. Relationship of Terms

3.1.3.1 Self-Defining Terms

A self-defining term, or constant, has its value inherent in the term. The assembler program does not assign a value to the term but does derive the value from the term.

Self-defining terms are used to specify immediate data, addresses, registers, and input/output information to the assembler program. Three types of self-defining terms are available: decimal, hexadecimal, and character (or string).

A decimal self-defining term is zero, or a decimal integer that does not begin with zero. For 16-bit data, the value range is 0 to 65,535 for an unsigned decimal integer and -32,768 to +32,767 for a signed decimal integer. For 8-bit data, the value range is 0 to 255 for an unsigned decimal integer and -128 to +127 for a signed decimal integer. It should be noted that having signed and unsigned data is just a coding convenience made available because some instructions treat data as signed values and others treat data as unsigned values. For example, in 8-bit data, a signed number such as -1_{10} or an unsigned number such as 255_{10} is converted to FF_{16} by the assembler. The microprocessor performs signed arithmetic except when executing decimal-add instructions.

Examples:

```
                    decimal constants
                   ~~~~~~~~~~~~~~~~~~
.BYTE               200, -100, 0, +32
.DBYTE              0, 40000, -3165
LDI                 0
XRI                 20
```

A hexadecimal self-defining term may be specified in either of two ways. The term may start with X', or the term may start with a leading zero. The range of hexadecimal numbers is 0 to $FFFF_{16}$ for 16-bit data and 0 to FF_{16} for 8-bit data.

Examples:

```
                    hexadecimal constants
                   ~~~~~~~~~~~~~~~~~~~~~
.BYTE               X'FF, X'10
.DBYTE              X'1FE, X'FFFF
XRI                 015
LDI                 X'40
```

A character self-defining term is defined as a string. A string is a series of characters or a single character enclosed in single quote marks (for example, 'THIS IS A STRING'). All letters, numbers, and special characters (including blanks) may be specified in a string. If a single quote mark is part of the character string, it should immediately be preceded by another single quote mark; for example, 'DON''T DO IT' represents DON'T DO IT. A null string ('') causes the assembler to generate a single blank. String characters are translated to ASCII code (see appendix A) in memory with each character occupying 8 bits. Refer to the .ASCII directive described in 4.5.9.

Examples:

```
                        ASCII constants
                        ~~~~~~~~~~~~~~~
.ASCII      'NUMBER='
XRI         'Y'
LDI         '?'
```

3.1.3.2 Symbolic Terms

Symbols are the most common means of referencing address locations or arbitrary values. Symbols are defined (assigned values) by one of three methods:

1. By appearing in a label field in a statement (see 3.2.1).

```
symbol
~~~~~~
SUB1:    LDI     0          ;CLEAR AC
```

   The value assigned to a symbol appearing in the label field is the address of the instruction, data, or storage location named by the symbol.

2. By using an assignment statement to assign a specific value to a symbol (see 4.4).

```
symbol
~~~~~~
P2 = 2                      ;STACK POINTER
```

3. By using a .FORM directive statement to assign a value to a symbol (see 4.5.12).

```
            symbol
            ~~~~~~
.FORM       DATA,2,2,4(X'A)
```

NOTE

The .FORM directive is not available in all assemblers.

Symbol construction must meet the following restrictions:

1. A symbol may contain one or more alphanumeric characters, the first of which must be either a letter or a dollar sign ($).

2. Although up to 32 characters may be included, only the first six characters are recognized by the assembler program. Therefore, the programmer must ensure that a long symbol is unique in the first six characters.

   Example:

```
LONGSY
LONGSYMBOL1   } are identical to the assembler
LONGSYMBOL2
```

3. If the first character in the symbol is a dollar sign ($), the symbol is defined as a local symbol. The . LOCAL operator allows the programmer to specify that local symbols appearing between two . LOCAL directive statements have a certain meaning only within that region of the program (see 4.5.10, . LOCAL Directive). This enables the programmer to use common mnemonics throughout a program without causing a conflict of names.

NOTE

Within a local region, a long local symbol must be unique in the first five characters, including the dollar sign ($).

Example: $ABCD, $ABCDEF } are identical symbols to the assembler.

4. No special characters or embedded blanks may appear within a symbol.

5. Symbol values cannot exceed 65,535 for unsigned 16-bit data, -32,768 to +32,767 for signed 16-bit data, 255 for unsigned 8-bit data or -128 to +127 for signed 8-bit data.

Several examples of symbols follow:

Legal Symbols	Illegal Symbols	Reason Illegal
$ABC	LONGSYMBOL1	First six characters are not unique
LONGSYMBOL	LONGSYMBOL2	
$AB2	2AB	First character must be a letter or a dollar sign
$2	#CDE	
XYZ	XYZ$	Last character is not alphanumeric
$ABCDEF	$ABCDE	First five characters of the local symbols are not unique.
$ABC2EF	$ABCDF	

A symbolic term may represent a memory address and, hence, may have a value ranging between 0 and $65,535_{10}$. Since SC/MP is an 8-bit machine, such a value requires two 8-bit bytes for its containment. In order to facilitate working with such values, they have been divided by the assembler into two halves, designated the "high" and the "low" parts of the value. The high part represents the upper half of the value (bits 15-8) and the low part represents the lower half (bits 7-0). These may be referred to in assembly language by using the forms, H(SYMBOL) and L(SYMBOL). For example:

If SYMBOL = X'F0D9,
then H(SYMBOL) = X'F0,
L(SYMBOL) = X'D9,
H(SYMBOL) + 1 = X'F1,
and L(SYMBOL) + 2 = X'DB.

Or if OTHER = SYMBOL + 3,
then OTHER = X'F0DC,
H(OTHER) = X'F0,
and L(OTHER) = X'DC.

The forms, H(SYMBOL) and L(SYMBOL), may be used in any context where an 8-bit value would be appropriate.

3.1.4 Expressions

Operand entries (see 3.2.3), consisting of either a single term or an arithmetic or a logical combination of terms, are called expressions. Expressions are either simple or multiterm. Simple expressions are single terms, such as a symbol or a self-defining term. Multiterm expressions are simple expressions combined using the arithmetic and logical operators shown in table 3-1. The multiterm expression is evaluated by the assembler program in a left-to-right order regardless of the operators used between the terms. Parentheses are not permitted for the purpose of grouping arithmetic and/or logical operations; they have special significance in defining certain assembler functions.

Examples:

```
L(TABLE) + X'10
100 - 1
ENTRY1 + ENTRY2 - 4
A > B * DISKAD
```

The result of the evaluated expression is a 16-bit value; however, the magnitude of the expression must be compatible with the storage allocated for the expression. For example, if the expression is to be stored in 8 bits, then the value of the expression must not exceed FF_{16}.

Example:

```
LDI     X'40 + CHAR          Expression value must not
                             exceed FF16.
```

If the expression is used to indicate a register, then the value must not exceed 3.

Example:

```
AND     0 (PTR + N)          Expression value must not
                             exceed 3.
```

Table 3-1 lists the arithmetic and logical operators available for forming expressions.

A unary operator operates upon one operand and appears in the format 'operator operand' (for example, -9). A binary operator operates upon two operands and appears in the format 'operand1 operator operand2' (for example, A&B).

The expressions "A < B", "A = B", and "A > B" cause the specified comparison to be made. The result is one if the condition is true or zero if the condition is false.

Table 3-1. Arithmetic and Logical Operators

Operator	Function	Type
+	Addition	Binary
-	Subtraction	Unary or binary
*	Multiplication	Binary
/	Division	Binary
%	Logical NOT	Unary
&	Logical AND	Binary
!	Logical OR	Binary
<	"Less Than"	Binary
=	"Equal To"	Binary
>	"Greater Than"	Binary

3.2 ASSEMBLER CODING CONVENTIONS

Assembly language programs consist of source statements, each of which occupies one line of text. There are five types of source statements accepted by the assembler: the instruction, the comment, the pseudo instruction, the assignment, and the directive. Except for comment statements, each type of source statement contains from one to five fields in the following order.

[label field] operation field [operand field] [comment field] [identification field]

A sample coding form with the five fields delineated is shown in figure 3-2. Since the assembler accepts free-form statements, the user may disregard field boundaries. However, for clarity and readability, the use of field boundaries is highly recommended.

3.2.1 Label Field

The label field is optional and may contain a symbol used to identify the current statement when referenced in other statements. More than one label may appear in the label field, in which case any of the labels may be used to reference the labeled location. A label may appear by itself in a statement, in which case it refers to the next instruction or data word in the source program.

Example:
```
LABEL1:
LABEL2:
LABEL3: LABEL4: LDI 1
```

3.2.2 Operation Field

The operation field is mandatory and contains a mnemonic operation code (opcode) that defines an assembler (such as a directive) or a machine operation (such as a load). A blank terminates the operation field.

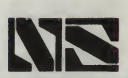
Operation mnemonics are used in directive and instruction statements. Instruction statements define the machine operations necessary to perform the desired function. Valid operation mnemonics for instruction statements are listed in appendix B. Directive statements control the process of program assembly and may generate data. Valid directive mnemonics are listed in appendix D.

3.2.3 Operand Field

The operand field contains entries that identify data to be acted upon by the statement. A space is not required to terminate the field. An operand entry is composed of one or more terms which represent a value. The value may be inherent in the term, in which case the term is self-defining (3.1.3.1); or the value may be assigned by the assembler program during assembly, in which case the term is symbolic (3.1.3.2). An arithmetic combination of terms is reduced to a single value by the assembler program as described in 3.1.4. The relationship of terms is shown in figure 3-1.

3.2.4 Comment Field

Comments are optional descriptive notes printed on the program listing for programmer reference. Comments should be included throughout the program to explain subroutine linkages, assumptions made, formats of inputs processed, and so forth. A comment may follow a statement on the same line, or the comment may be entered on one or more separate statement lines. The comment has no affect on the assembled program, but it is printed on the program listing.



CODING FORM

PROGRAM	PUNCHING INSTRUCTIONS	GRAPHIC										PAGE OF
PROGRAMMER		PUNCH										DATE

STATEMENT

LABEL 1 2 3 4 5 6 7 8	OPERATION 9 10 11 12 13 14 15 16	OPERAND 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32	COMMENTS 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	IDENTIFICATION 73 74 75 76 77 78 79 80
1 2 3 4 5 6 7 8	9 10 11 12 13 14 15 16	17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32	33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80

Figure 3-2. Sample Coding Form

The following conventions apply to comments:

1. A comment must be preceded by a semicolon (;).
2. All valid characters, including blanks, may be used in comments.
3. Comments should not extend beyond column 72, but a comment may be carried over on the following line (preceded by a semicolon).

3.2.5 Identification Sequence Field

The identification sequence field is an optional entry that specifies program identification and/or statement sequence characters. If the field or a portion of the field is used for program identification, the identification is punched in the statement cards and is printed on the program listing. This field generally is not used with paper tape input.

As an aid to keeping source statements in order, the programmer may code a sequence of characters in ascending order in the identification sequence field.

The identification sequence field is fixed in columns 73 through 80 of the source image. Columns 73 through 80 are ignored by the assembler but are printed in the program listing.

In some cases, the width of the carriage of the printer is not wide enough to include the identification sequence field in the program listing.

3.2.6 Example of Statement

An example of an assembler statement follows:

Label	Operation	Operand	Comment
GETVAL:	LD	1(P2)	;GET A VALUE

The label, GETVAL, is a symbolic name for the address of this instruction. Thus, GETVAL can be used in other statements (preceeding or following) to address this statement. The mnemonic operation code, LD, stipulates the type of operation. The operand field specifies a pointer, P2, and a displacement, +1; and the comment field contains a note that may be used by the programmer to identify quickly the action defined by the statement. See chapter 5 for other statement examples.

Chapter 4

STATEMENTS

The SC/MP assembler accepts five types of statements: instruction, comment, pseudo instruction, assignment, and directive. One of these basic statements or a combination of them may be used to create macros. The individual statement types are described in this chapter. The definition and the use of macros is described in chapter 6.

4.1 INSTRUCTION STATEMENTS

The assembly language instruction set of the SC/MP provides arithmetic, logic, shift, transfer, and other operations between the accumulator and memory or the other registers.

Instruction statements, when assembled, generate the object (machine) code that defines the operations the processor will perform. Depending on the instruction type, one or two bytes of object code are generated for each instruction assembled.

In the following descriptions, any user accessible register or bit that is not explicitly mentioned will not be altered by the instruction.

There are 46 SC/MP instruction statements that comprise the following eight classes:

- Memory Reference
- Memory Increment/Decrement
- Immediate
- Transfer
- Extension Register
- Pointer Register Move
- Shift, Rotate, and Serial Input/Output
- Miscellaneous

Refer to table 4-1 for definitions of the symbols used in the notation for describing the SC/MP instruction set. Upper-case mnemonics refer to units designated by fields of the instruction words; lower-case mnemonics refer to the numerical values of the corresponding fields. For example, ptr in an assembler statement denotes the number of a pointer register, whereas (AC) ← (PTR) denotes the contents of the accumulator are replaced by the contents of a pointer register. In the case where both a lower-case mnemonic and an upper-case mnemonic are composed of the same letters, only the lower-case mnemonic is given in table 4-1. Lower-case notation designates a variable.

The SC/MP instruction set is summarized in table 4-2.

Alternate opcodes are given for some instructions. The alternates listed are consistent with those used for other microprocessors manufactured by National Semiconductor.

Table 4-1. Symbols and Notation

Symbol and Notation	Meaning
AC	8-bit Accumulator.
address	In the operand field of memory-reference, memory increment/decrement or transfer instruction source statements, address indicates the symbolic expression that represents a memory location. The assembler calculates the displacement (disp) for the instruction from address and generates the instruction with PC-relative addressing.
CY/L	Carry/Link Flag in the Status Register.
data	8-bit immediate data field. Data may represent a signed or an unsigned twos complement number or a binary coded decimal (BCD) number.
disp	Displacement represents a signed 8-bit address modifier in a memory-reference, memory increment/decrement or transfer instruction.
EA	Effective Address as specified by the instruction.
E	Extension Register; provides for temporary storage, variable displacements and separate serial input/output port.
i	Represents the bit in one of the bit positions, 7 through 1, of the Accumulator or the Extension Register.
IE	Interrupt Enable Flag.
m	Mode bit, used in memory reference instructions. Blank parameter sets m = 0, @ sets m = 1.
OV	Overflow Flag in the Status Register.
PC	Program Counter (Pointer Register 0); during address formation, PC points to the last byte of the instruction being executed.
ptr	Pointer Register (ptr = 0 through 3). The register specified in byte 1 of the instruction.
$ptr_{n:m}$	Pointer register bits; n:m = 7 through 0 or 15 through 8.
SIN	Serial Input pin.
SOUT	Serial Output pin.
SR	8-bit Status Register.
( )	Means "contents of." For example, (EA) is contents of Effective Address.
[ ]	Means optional field in the assembler instruction format.
~	Ones complement of value immediately to right of ~ .
→	"Replaces".
←	"Is replaced by".
↔	"Exchange".
@	When used in the operand field of the instruction, sets the mode bit (m) to 1 for auto-incrementing/auto-decrementing indexing.
10^+	Modulo 10 addition.
∧	AND operation.
∨	Inclusive-OR operation.
▽	Exclusive-OR operation.
≥	Greater than or equal to.
=	Equals.
≠	Does not equal.

Table 4-2. SC/MP Instruction Summary --- Double-Byte

Description	Opcode Base	Source Statement	Instruction Format	Operation	Micro-cycles	Page
Memory Reference Instructions						
Load	C0XX	LD	1 1 0 0 0 m ptr disp	(AC)←(EA)	18	4-5
Store	C8XX	ST	1 1 0 0 1 m ptr disp	(EA)←(AC)	18	4-6
AND	D0XX	AND	1 1 0 1 0 m ptr disp	(AC)←(AC) ∧ (EA)	18	4-6
OR	D8XX	OR	1 1 0 1 1 m ptr disp	(AC)←(AC) ∨ (EA)	18	4-6
Exclusive-OR	E0XX	XOR	1 1 1 0 0 m ptr disp	(AC)←(AC) ▽ (EA)	18	4-6
Decimal Add	E8XX	DAD, DECA	1 1 1 0 1 m ptr disp	(AC)←(AC)$_{10}$ + (EA)$_{10}$ + CY/L ; CY/L	23	4-7
Add	F0XX	ADD	1 1 1 1 0 m ptr disp	(AC)←(AC) + (EA) + CY/L ; CY/L , OV	19	4-7
Complement and Add	F8XX	CAD	1 1 1 1 1 m ptr disp	(AC)←(AC) + ~ (EA) + CY/L ; CY/L , OV	20	4-8
		address / disp(ptr) / @disp(ptr)				
Memory Increment/Decrement Instructions						
Increment and Load	A8XX	ILD	1 0 1 0 1 0 ptr disp	(AC), (EA) ←(EA) + 1	22	4-8
Decrement and Load	B8XX	DLD	1 0 1 1 1 0 ptr disp	(AC), (EA)←(EA) - 1	22	4-9
		address / disp(ptr)				
Immediate Instructions						
Load Immediate	C4XX	LDI, LI	1 1 0 0 0 1 0 0 data	(AC)←data	10	4-9
AND Immediate	D4XX	ANI	1 1 0 1 0 1 0 0 data	(AC)←(AC) ∧ data	10	4-10
OR Immediate	DCXX	ORI	1 1 0 1 1 1 0 0 data	(AC)←(AC) ∨ data	10	4-10
Exclusive OR Immediate	E4XX	XRI	1 1 1 0 0 1 0 0 data	(AC)←(AC) ▽ data	10	4-10
Decimal Add Immediate	ECXX	DAI	1 1 1 0 1 1 0 0 data	(AC)←(AC)$_{10}$ + data$_{10}$ + CY/L ; CY/L	15	4-11
Add Immediate	F4XX	ADI	1 1 1 1 0 1 0 0 data	(AC)←(AC) + data + CY/L ; CY/L , OV	11	4-11
Complement and Add Immediate	FCXX	CAI	1 1 1 1 1 1 0 0 data	(AC)←(AC) + ~ data + CY/L ; CY/L , OV	12	4-11
		data				
Transfer Instructions						
Jump	90XX	JMP	1 0 0 1 0 0 ptr disp	(PC)←EA	11	4-12
Jump If Positive	94XX	JP	1 0 0 1 0 1 ptr disp	If (AC) ≥ 0, (PC)←EA	9,11	4-13
Jump If Zero	98XX	JZ	1 0 0 1 1 0 ptr disp	If (AC) = 0, (PC)←EA	9,11	4-13
Jump If Not Zero	9CXX	JNZ	1 0 0 1 1 1 ptr disp	If (AC) ≠ 0, (PC)←EA	9,11	4-13
		address / disp(ptr)				
Double-byte Miscellaneous Instructions						
Delay	8FXX	DLY data	1 0 0 0 1 1 1 1 data	count AC to -1, delay = 13 + 2 (AC) + 2 data + 2^9 data microcycles	13 to 131593	4-22

NOTES: 1. In the opcode base, XX = disp or data.

2. Where alternate mnemonics are listed in the source statement (for example, LDI and LI), only one should be used.

(Continued on next page)

Table 4-2. SC/MP Instruction Summary --- Single-Byte

Description	Opcode Base	Source Statement	Instruction Format	Operation	Micro-cycles	Page
Extension Register Instructions			7 6 5 4 3 2 1 0			
Load AC from Extension	40	LDE	0 1 0 0 0 0 0 0	(AC)←(E)	6	4-14
Exchange AC and Extension	01	XAE	0 0 0 0 0 0 0 1	(AC)←→(E)	7	4-14
AND AC with Extension	50	ANE	0 1 0 1 0 0 0 0	(AC)←(AC) ∧ (E)	6	4-14
OR AC with Extension	58	ORE	0 1 0 1 1 0 0 0	(AC)←(AC) ∨ (E)	6	4-14
Exclusive-OR AC with Extension	60	XRE	0 1 1 0 0 0 0 0	(AC)←(AC) ▽ (E)	6	4-15
Decimal Add AC and Extension	68	DAE	0 1 1 0 1 0 0 0	$(AC)_{10} \leftarrow (AC)_{10} + (E)_{10} + CY/L$; CY/L	11	4-15
Add AC and Extension	70	ADE	0 1 1 1 0 0 0 0	(AC)←(AC) + (E) + CY/L ; CY/L , OV	7	4-15
Complement and Add Extension to AC	78	CAE	0 1 1 1 1 0 0 0	(AC)←(AC) + ~ (E) + CY/L ; CY/L , OV	8	4-16
Pointer Register Move Instructions			7 6 5 4 3 2 1 0			
Exchange Pointer Low	30	XPAL	0 0 1 1 0 0	(AC)←→$(PTR_{7:0})$	8	4-16
Exchange Pointer High	34	XPAH ptr	0 0 1 1 0 1 ptr	(AC)←→$(PTR_{15:8})$	8	4-17
Exchange Pointer with PC	3C	XPPC	0 0 1 1 1 1	(PC)←→(PTR)	7	4-17
Shift, Rotate, Serial I/O Instructions			7 6 5 4 3 2 1 0			
Serial Input/Output	19	SIO	0 0 0 1 1 0 0 1	$(E_i) \rightarrow (E_{i-1})$, SIN→$(E_7)$, (E_0)→SOUT	5	4-17
Shift Right	1C	SR, SHR	0 0 0 1 1 1 0 0	$(AC_i) \rightarrow (AC_{i-1})$, 0→$(AC_7)$	5	4-18
Shift Right with Link	1D	SRL	0 0 0 1 1 1 0 1	$(AC_i) \rightarrow (AC_{i-1})$, CY/L →$(AC_7)$	5	4-18
Rotate Right	1E	RR, ROR	0 0 0 1 1 1 1 0	$(AC_i) \rightarrow (AC_{i-1})$, $(AC_0) \rightarrow (AC_7)$	5	4-18
Rotate Right with Link	1F	RRL	0 0 0 1 1 1 1 1	$(AC_i) \rightarrow (AC_{i-1})$, (AC_0)→ CY/L →(AC_7)	5	4-19
Single-byte Miscellaneous Instructions			7 6 5 4 3 2 1 0			
Halt	00	HALT	0 0 0 0 0 0 0 0	Pulse H-flag	8	4-19
Clear Carry/Link	02	CCL	0 0 0 0 0 0 1 0	CY/L ← 0	5	4-20
Set Carry/Link	03	SCL	0 0 0 0 0 0 1 1	CY/L ← 1	5	4-20
Disable Interrupt	04	DINT	0 0 0 0 0 1 0 0	IE ← 0	6	4-21
Enable Interrupt	05	IEN	0 0 0 0 0 1 0 1	IE ← 1	6	4-20
Copy Status to AC	06	CSA	0 0 0 0 0 1 1 0	(AC)←(SR)	5	4-21
Copy AC to Status	07	CAS	0 0 0 0 0 1 1 1	(SR)←(AC)	6	4-21
No Operation	08	NOP	0 0 0 0 1 0 0 0	(PC)←(PC) + 1	5	4-22

NOTE: Where alternate mnemonics are listed in the source statement (for example, SR, SHR), only one should be used.

4.1.1 Memory Reference Instructions

This group of eight instructions provides logic, arithmetic, and data-transfer operations between the accumulator and the effective address. The Memory Reference Instructions and mnemonics are as follows:

Load	LD
Store	ST
AND	AND
OR	OR
Exclusive-OR	XOR
Decimal Add	DAD, DECA
Add	ADD
Complement and Add	CAD

The Effective Address (EA) may be PC-relative, indexed, or auto-indexed as shown in table 4-3.

Table 4-3. Memory Reference Formats

Addressing	Operand Formats			
	Object			Source
	m	ptr	displacement	
PC-relative	0	0	-128* to +127	address
Indexed	0	1, 2, or 3	-128* to +127	disp(ptr)
Auto-indexing	1	1, 2, or 3	-128* to +127	@disp(ptr)

*Note: The contents of the Extension Register are substituted for the displacement if the displacement equals -128_{10} (80_{16}).

PC-relative addressing is assumed when only a symbolic name of an address is specified (example: LD VALUE). Indexed addressing requires a displacement and one of the four pointer registers (example: ADD 10(P1)). Auto-indexing requires the "at sign," a displacement, and a pointer register (other than PC) (example: ST @-1(3)).

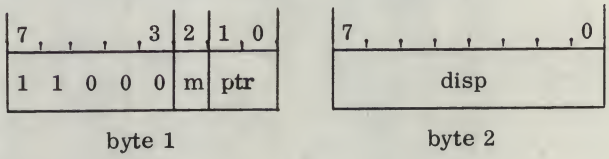
The source statement format, the instruction format, and the description of the operation of each Memory Reference Instruction follow.

LOAD (LD)

SOURCE STATEMENT

Mnemonic	Operands
LD	address disp(ptr) @disp(ptr)

INSTRUCTION FORMAT



Operation: (AC) ← (EA)

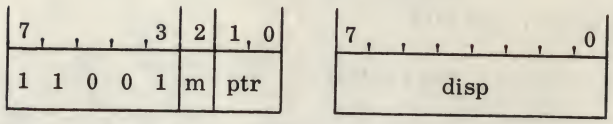
The contents of the Effective Address, (EA), replace the contents of the Accumulator, (AC). The initial contents of AC are lost; the contents of EA are not altered.

STORE (ST)

SOURCE STATEMENT

Mnemonic	Operands
ST	address disp(ptr) @disp(ptr)

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (EA) ← (AC)

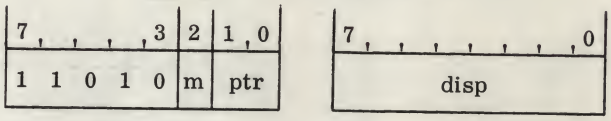
The contents of the Accumulator, (AC), replace the contents of the Effective Address, (EA). The initial contents of EA are lost; the contents of AC are not altered.

AND (AND)

SOURCE STATEMENT

Mnemonic	Operands
AND	address disp(ptr) @disp(ptr)

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC) ← (AC) ∧ (EA)

The contents of the Accumulator, (AC), are ANDed with the contents of the Effective Address, (EA), and the result is stored in AC. The initial contents of AC are lost; the contents of EA are not altered. The truth table for this instruction is shown below.

AC_b	E_b	$AC_b \wedge E_b$
0	0	0
0	1	0
1	0	0
1	1	1

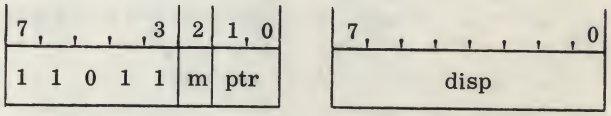
b = bits 7 to 0

OR (OR)

SOURCE STATEMENT

Mnemonic	Operands
OR	address disp(ptr) @disp(ptr)

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC) ← (AC) ∨ (EA)

The contents of the Accumulator, (AC), are inclusive-ORed with the contents of the Effective Address, (EA), and the result is stored in AC. The initial contents of AC are lost; the contents of EA are not altered. The truth table for this instruction is shown below.

AC_b	E_b	$AC_b \wedge E_b$
0	0	0
0	1	1
1	0	1
1	1	1

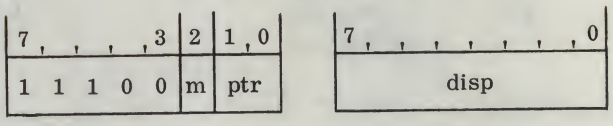
b = bits 7 to 0

EXCLUSIVE-OR (XOR)

SOURCE STATEMENT

Mnemonic	Operands
XOR	address disp(ptr) @disp(ptr)

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC) ← (AC) ▽ (EA)

The contents of the Accumulator, (AC), are exclusive-ORed with the contents of the Effective Address, (EA), and the result is stored in AC. The initial contents of AC are lost; the contents of EA are not altered. The truth table for this instruction is shown below.

AC_b	E_b	$AC_b \triangledown E_b$
0	0	0
0	1	1
1	0	1
1	1	0

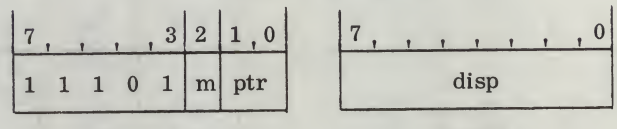
b = bits 7 to 0

DECIMAL ADD (DAD, DECA)

SOURCE STATEMENT

Mnemonic	Operands
DAD DECA	address disp(ptr) @disp(ptr)

INSTRUCTION FORMAT

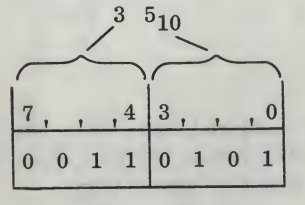


byte 1 byte 2

Operation: (AC) ← $(AC)_{10} + (EA)_{10}$ + CY/L; CY/L

The contents of the Accumulator, (AC), and the contents of the Effective Address, (EA), are treated as 2-digit binary-coded-decimal (BCD) numbers. The contents of AC, the contents of EA, and the Carry Flag, CY/L, are added, and the 2-digit BCD sum is stored in AC. The initial contents of AC are lost; the contents of EA are not altered. The Carry Flag is set if a carry occurs from the most significant decimal digit; otherwise, it is cleared. The Overflow Flag is not affected.

Example of a 2-digit BCD number:



(AC) or (EA)

NOTE

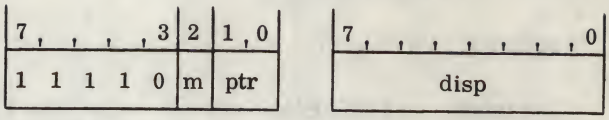
The range of a 2-digit BCD number is 0 through 99_{10} ($0 \leqslant BCD \leqslant 99$).

ADD (ADD)

SOURCE STATEMENT

Mnemonic	Operands
ADD	address disp(ptr) @disp(ptr)

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC) ← (AC) + (EA) + CY/L ; CY/L, OV

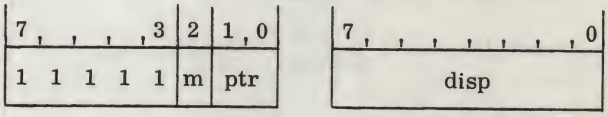
The contents of the Accumulator, (AC), and the contents of the Effective Address, (EA), are treated as 8-bit binary twos complement numbers. The contents of AC, the contents of EA, and the Carry Flag, CY/L, are added, and the sum is stored in AC. The initial contents of AC are lost; the contents of EA are not altered. The Carry Flag is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag, OV, is set if the sign of the result differs from the sign of both operands; otherwise, it is cleared.

COMPLEMENT AND ADD (CAD)

SOURCE STATEMENT

Mnemonic	Operands
CAD	address disp(ptr) @disp(ptr)

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC) ← (AC) + ~(EA) + CY/L; CY/L, OV

The contents of the Accumulator, (AC), and the contents of the Effective Address, (EA), are treated as 8-bit binary, twos complement numbers. The contents of AC, the ones complement of the contents of EA, and the Carry Flag, CY/L, are added, and the sum is stored in AC. The initial contents of AC are lost; the contents of EA are not altered. The Carry Flag is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag, OV, is set if the sign of the result is the same as the sign of the contents of EA and opposite the sign of the contents of AC; otherwise OV is cleared.

NOTE

If the Carry Flag is cleared initially, the ones complement of the contents of E is added to the contents of AC. If the Carry Flag is set, the twos complement of the contents of E is added to the contents of AC.

4.1.2 Memory Increment/Decrement Instructions

The two double-byte instructions in this group may be used to maintain a software counter in memory. The Memory Increment/Decrement Instructions and mnemonics are as follows:

Increment and Load . . . . . ILD
Decrement and Load . . . . . DLD

The source statement format, the instruction format, and the description of each Memory Increment/Decrement Instruction follow.

NOTE

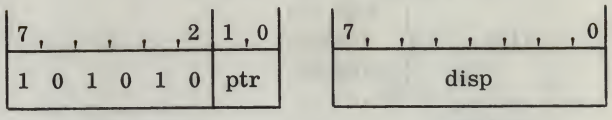
At the hardware level, these instructions access the memory in a read-alter-write mode. The processor retains control of the input/output bus between the data read and write operations.

INCREMENT AND LOAD (ILD)

SOURCE STATEMENT

Mnemonic	Operands
ILD	{ address / disp(ptr)

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC), (EA) ← (EA) + 1

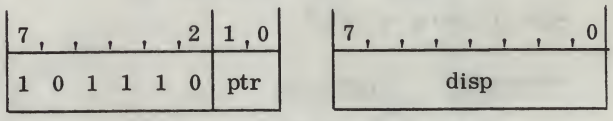
The contents of the Effective Address, (EA), are incremented by 1, and the result is stored in the Accumulator, AC, and, also, in EA. The initial contents of AC and EA are lost.

DECREMENT AND LOAD (DLD)

SOURCE STATEMENT

Mnemonic	Operands
DLD	{ address / disp(ptr)

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC), (EA) ← (EA) - 1

The contents of the Effective Address, (EA), are decremented by 1, and the result is stored in the Accumulator, AC, and, also, in EA. The initial contents of AC and EA are lost.

4.1.3 Immediate Instructions

The immediate instructions perform most of the same operations as the memory-reference instructions. The data used in the operations comes from the byte immediately after the opcode byte; that is, the data byte is the displacement. The Immediate Instructions and mnemonics are as follows:

Load Immediate	LDI, LI
AND Immediate	ANI
OR Immediate	ORI
Exclusive-OR Immediate	XRI
Decimal Add Immediate	DAI
Add Immediate	ADI
Complement and Add Immediate	CAI

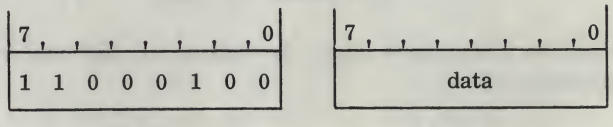
The source statement format, the instruction format, and the description of each Immediate Instruction follow.

LOAD IMMEDIATE (LDI, LI)

SOURCE STATEMENT

Mnemonic	Operand
LDI / LI	data

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC) ← data

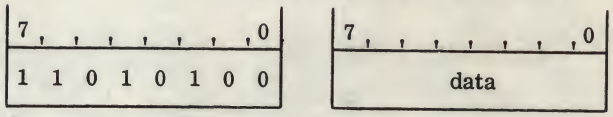
The data byte replaces the contents of the Accumulator, (AC). The initial contents of AC are lost; the data byte is not altered.

AND IMMEDIATE (ANI)

SOURCE STATEMENT

Mnemonic	Operand
ANI	data

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC) ← (AC) ∧ data

The contents of the Accumulator, (AC), are ANDed with the data byte, and the result is stored in AC. The initial contents of AC are lost; the data byte is not altered. The truth table for this instruction is shown below.

AC_b	$data_b$	$AC_b \wedge data_b$
0	0	0
0	1	0
1	0	0
1	1	1

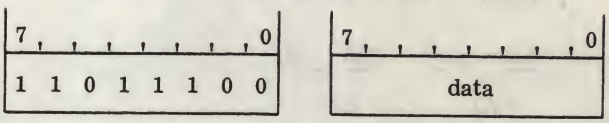
b = bits 7 to 0

OR IMMEDIATE (ORI)

SOURCE STATEMENT

Mnemonic	Operand
ORI	data

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC) ← (AC) ∨ data

The contents of the Accumulator, (AC), are inclusive-ORed with the data byte, and the result is stored in AC. The initial contents of AC are lost; the data byte is not altered. The truth table for this instruction is shown below.

AC_b	$data_b$	$AC_b \vee data_b$
0	0	0
0	1	1
1	0	1
1	1	1

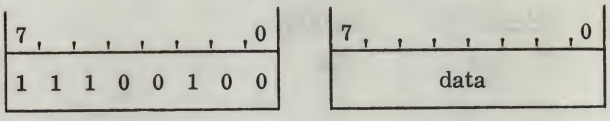
b = bits 7 to 0

EXCLUSIVE-OR IMMEDIATE (XRI)

SOURCE STATEMENT

Mnemonic	Operand
XRI	data

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC) ← (AC) ▽ data

The contents of the Accumulator, (AC), are exclusive-ORed with the data byte, and the result is stored in AC. The initial contents of AC are lost; the data byte is not altered. The truth table for this instruction is shown below.

AC_b	$data_b$	$AC_b \triangledown data_b$
0	0	0
0	1	1
1	0	1
1	1	0

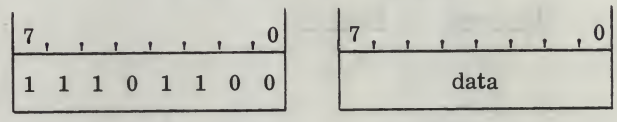
b = bits 7 to 0

DECIMAL ADD IMMEDIATE (DAI)

SOURCE STATEMENT

Mnemonic	Operand
DAI	data

INSTRUCTION FORMAT



byte 1 byte 2

Operation: (AC) ← $(AC)_{10}$ + $data_{10}$ + CY/L; CY/L

The contents of the Accumulator, (AC), and the data byte are treated as 2-digit binary-coded-decimal (BCD) numbers. The contents of AC, the data byte, and the Carry, CY/L, are added, and the 2-digit BCD sum is stored in AC. The initial contents of AC are lost; the data byte is not altered. The Carry Flag is set if a carry from the most significant decimal digit occurs; otherwise, it is cleared. The Overflow Flag is not affected.

NOTE

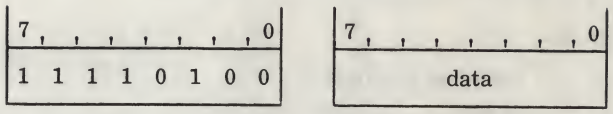
To code a BCD number in the data field of the source statement, the user must enter it as a hexadecimal value. Example: 058 or X'58 to represent BCD 58.

ADD IMMEDIATE (ADI)

SOURCE STATEMENT

Mnemonic	Operand
ADI	data

INSTRUCTION FORMAT



Operation: (AC) ← (AC) + data + CY/L; CY/L, OV

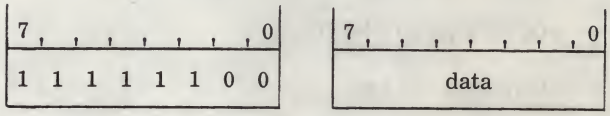
The contents of the Accumulator, (AC), and the data byte are treated as 8-bit binary, twos complement numbers. The contents of AC, the data byte, and the Carry Flag, CY/L, are added, and the sum is stored in AC. The initial contents of AC are lost; the data byte is not altered. The Carry Flag is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag, OV, is set if the sign of the result differs from the sign of both operands; otherwise, it is cleared.

COMPLEMENT AND ADD IMMEDIATE (CAI)

SOURCE STATEMENT

Mnemonic	Operand
CAI	data

INSTRUCTION FORMAT



Operation: (AC) ← (AC) + ~ data + CY/L; CY/L, OV

The contents of the Accumulator, (AC), and the data byte are treated as 8-bit binary, twos complement numbers. The contents of AC, the ones complement of the data byte, and the Carry Flag, CY/L, are added, and the result is stored in AC. The initial contents of AC are lost; the data byte is not altered. The Carry Flag is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag, OV, is set if the sign of the result is the same as the sign of the data byte and opposite the sign of the contents of AC; otherwise, it is cleared.

NOTE

If the Carry Flag is set initially, this operation is equivalent to subtracting the data byte from the contents of AC.

4.1.4 Transfer Instructions

The four double-byte instructions in this group are used for conditional and unconditional jumps within a routine, and jumps to subroutines. The Transfer Instructions and mnemonics are as follows:

Jump	JMP
Jump if Positive	JP
Jump if Zero	JZ
Jump if Not Zero	JNZ

The effective address of a jump that is PC-relative is the PC plus the displacement (disp). The range of a PC-relative jump is -126 to +129 bytes from the jump instruction. As with the memory-reference instructions, the effective address does not affect the 4 most significant address bits; thus, wrap-around can occur at 4K page boundaries. When 'ptr' specifies a pointer register other than 0, the 4 most significant bits of the PC are replaced by the 4 most significant bits of the specified pointer.

When assembling Transfer Instructions, the assembler reduces the specified displacement by one (1), so if the jump is taken the next instruction executed is located at the label specified in the jump instruction; that is, the effective address specified by the user.

Example:

```
Location Counter   Object Code
      1000            9012      START:   JMP     LABEL     ;JUMP TO LABEL
                                           .
                                           .
                                           .
      1014                      LABEL:
PC (during execution)           1001
displacement                     12
EA                              1013
```

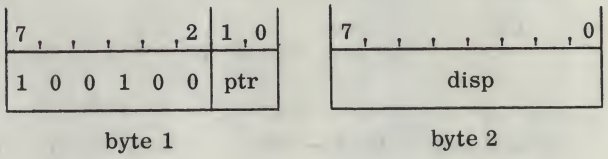
The source statement format, the instruction format, and the description of the operation of each Transfer Instruction follow.

JUMP (JMP)

SOURCE STATEMENT

Mnemonic	Operands
JMP	{ address / disp(ptr)

INSTRUCTION FORMAT



Operation: (PC) ← EA

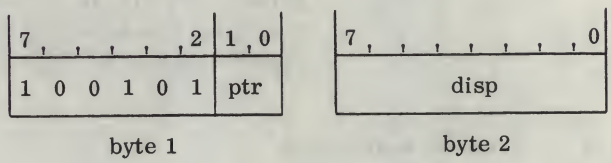
The Effective Address, EA, replaces the contents of the Program Counter, (PC). The next instruction is fetched from the location designated by the new contents of PC + 1.

JUMP IF POSITIVE (JP)

SOURCE STATEMENT

Mnemonic	Operands
JP	{ address / disp(ptr)

INSTRUCTION FORMAT



Operation: If (AC) $\geq$ 0, (PC) ← EA

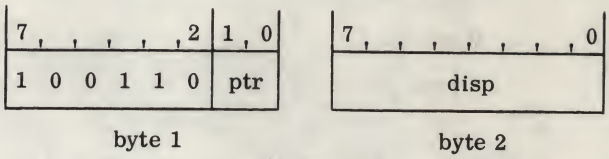
If the contents of the Accumulator, (AC), are positive or zero, the Effective Address, EA, replaces the contents of the Program Counter, (PC). The next instruction is fetched from the location designated by the new contents of PC + 1.

JUMP IF ZERO (JZ)

SOURCE STATEMENT

Mnemonic	Operands
JZ	{ address disp(ptr)

INSTRUCTION FORMAT



Operation: If (AC) = 0, (PC) ← EA

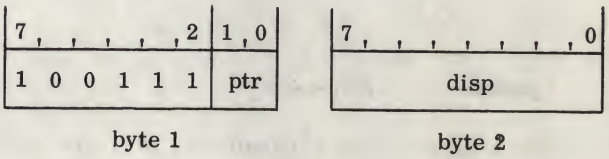
If the contents of the Accumulator, (AC), are zero, the Effective Address, EA, replaces the contents of the Program Counter, (PC). The next instruction is fetched from the location designated by the new contents of the PC + 1.

JUMP IF NOT ZERO (JNZ)

SOURCE STATEMENT

Mnemonic	Operands
JNZ	{ address disp(ptr)

INSTRUCTION FORMAT



Operation: If (AC) ≠ 0, (PC) ← EA

If the contents of the Accumulator, (AC), are not zero, the Effective Address, EA, replaces the contents of the Program Counter, (PC). The next instruction is fetched from the location designated by the new contents of the PC + 1.

4.1.5 Extension Register Instructions

This group of eight single-byte instructions is used for arithmetic and logic operations between the Extension Register (E) and the Accumulator (AC). The Extension Register Instructions and mnemonics are as follows:

Load AC from Extension	LDE
Exchange AC and Extension	XAE
AND AC with Extension	ANE
OR AC with Extension	ORE
Exclusive-OR AC with Extension	XRE
Decimal Add AC and Extension	DAE
Add AC and Extension	ADE
Complement and Add Extension to AC	CAE

The source statement format, the instruction format, and the description of the operation of each Extension Register Instruction follow.

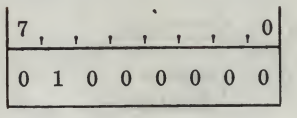
LOAD AC FROM EXTENSION (LDE)

SOURCE STATEMENT

Mnemonic

LDE

INSTRUCTION FORMAT



Operation: (AC) ← (E)

The contents of the Extension Register, (E), replace the contents of the Accumulator, (AC). The initial contents of AC are lost; the contents of E are not altered.

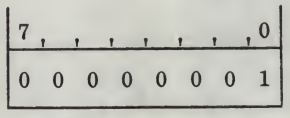
EXCHANGE AC AND EXTENSION (XAE)

SOURCE STATEMENT

Mnemonic

XAE

INSTRUCTION FORMAT



Operation: (AC) ←→ (E)

The contents of the Accumulator, (AC), are exchanged with the contents of the Extension Register, (E).

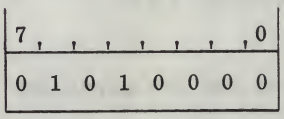
AND AC WITH EXTENSION (ANE)

SOURCE STATEMENT

Mnemonic

ANE

INSTRUCTION FORMAT



Operation: (AC) ← (AC) ∧ (E)

The contents of the Accumulator, (AC), are ANDed with the contents of the Extension Register, (E), and the result is stored in AC. The initial contents of AC are lost; the contents of E are not altered. The truth table for this instruction is shown below.

AC_b	E_b	$AC_b \wedge E_b$
0	0	0
0	1	0
1	0	0
1	1	1

b = bits 7 to 0

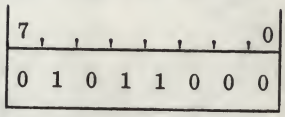
OR AC WITH EXTENSION (ORE)

SOURCE STATEMENT

Mnemonic

ORE

INSTRUCTION FORMAT



Operation: (AC) ← (AC) ∨ (E)

The contents of the Accumulator, (AC), are inclusive-ORed with the contents of the Extension Register, (E), and the result is stored in AC. The initial contents of AC are lost; the contents of E are not altered. The truth table for this instruction is shown below.

OR

AC_b	E_b	$AC_b \vee E_b$
0	0	0
0	1	1
1	0	1
1	1	1

b = bits 7 to 0

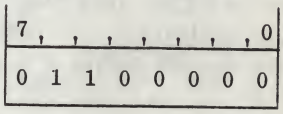
EXCLUSIVE-OR AC WITH EXTENSION (XRE)

SOURCE STATEMENT

Mnemonic

XRE

INSTRUCTION FORMAT



Operation: (AC) ← (AC) $\bar{\vee}$ (E)

The contents of the Accumulator, (AC), are exclusive-ORed with the contents of the Extension Register, (E), and the result is stored in AC. The initial contents of AC are lost; the contents of E are not altered. The truth table for this instruction is shown below.

AC_b	E_b	$AC_b \bar{\vee} E_b$
0	0	0
0	1	1
1	0	1
1	1	0

b = bits 7 to 0

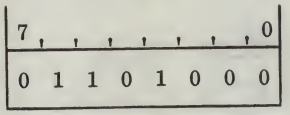
DECIMAL ADD AC AND EXTENSION (DAE)

SOURCE STATEMENT

Mnemonic

DAE

INSTRUCTION FORMAT



Operation: $(AC) \leftarrow (AC)_{10} + (E)_{10} + CY/L;\ CY/L$

The contents of the Accumulator, (AC), and the contents of the Extension Register, (E), are treated as 2-digit binary-coded-decimal (BCD) numbers. The contents of AC, the contents of E, and the Carry Flag, CY/L, are added, and the sum is stored in AC. The initial contents of AC are lost; the contents of E are not altered. The Carry Flag is set if a carry from the most significant decimal digit occurs; otherwise, it is cleared. The Overflow Flag is not affected.

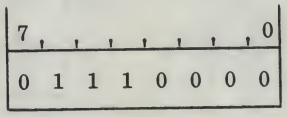
ADD AC AND EXTENSION (ADE)

SOURCE STATEMENT

Mnemonic

ADE

INSTRUCTION FORMAT



Operation: $(AC) \leftarrow (AC) + (E) + CY/L;\ CY/L,\ OV$

The contents of the Accumulator, (AC), and the contents of the Extension Register, (E), are treated as 8-bit binary, twos complement numbers. The contents of AC, the contents of E, and the Carry Flag, CY/L, are added, and the sum is stored in AC. The initial contents of AC are lost; the contents of E are not altered. The Carry Flag, CY/L, is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag, OV, is set if the sign of the result differs from the sign of both operands; otherwise, it is cleared.

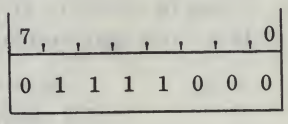
COMPLEMENT AND ADD EXTENSION TO AC (CAE)

SOURCE STATEMENT

Mnemonic

CAE

INSTRUCTION FORMAT



Operation: $(AC) \leftarrow (AC) + \sim (E) + CY/L;\ CY/L,\ OV$

The contents of the Accumulator, (AC), and the contents of the Extension Register, (E), are treated as 8-bit binary, twos complement numbers. The contents of AC, the ones complement of the contents of E, and the Carry Flag, CY/L, are added, and the result is stored in AC. The initial contents of AC are lost; the contents of E are not altered. The Carry Flag, CY/L, is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag, OV, is set if the sign of the result is the same as the sign of the contents of E and opposite the sign of the contents of AC; otherwise, it is cleared.

NOTE

If the Carry Flag is set initially, this operation is equivalent to subtracting the contents of E from the contents of AC.

4.1.6 Pointer Register Move Instructions

The three single-byte instructions in this group are used for transfers between the Pointer Registers and the Accumulator or the Program Counter. The Pointer Register Move Instructions and mnemonics are as follows:

Exchange Pointer Low	XPAL
Exchange Pointer High	XPAH
Exchange Pointer with PC	XPPC

The source statement format, the instruction format, and the description of each Pointer Register Move Instruction follow.

EXCHANGE POINTER LOW (XPAL)

SOURCE STATEMENT

Mnemonic	Operand
XPAL	ptr

INSTRUCTION FORMAT

7 2	1, 0
0 0 1 1 0 0	ptr

Operation: $(AC) \longleftrightarrow (PTR_{7:0})$

The contents of the Accumulator, (AC), are exchanged with the contents of the low-order byte (bits 7 through 0) of the designated Pointer Register, (PTR).

EXCHANGE POINTER HIGH (XPAH)

SOURCE STATEMENT

Mnemonic	Operand
XPAH	ptr

INSTRUCTION FORMAT

7 2	1, 0
0 0 1 1 0 1	ptr

Operation: $(AC) \longleftrightarrow (PTR_{15:8})$

The contents of the Accumulator, (AC), are exchanged with the contents of the high-order byte (bits 15 through 8) of the designated Pointer Register, (PTR).

EXCHANGE POINTER WITH PC (XPPC)

SOURCE STATEMENT

Mnemonic	Operand
XPPC	ptr

INSTRUCTION FORMAT

7 2	1, 0
0 0 1 1 1 1	ptr

Operation: $(PC) \longleftrightarrow (PTR)$

The contents of the Program Counter, (PC), are exchanged with the contents of the designated Pointer Register, (PTR).

4.1.7 Shift, Rotate, Serial Input/Output Instructions

The five single-byte instructions in this group shift or rotate the Accumulator or perform serial input/output operations using the Extension Register. The Shift, Rotate, and Serial Input/Output Instructions and mnemonics are as follows:

Serial Input/Output	SIO
Shift Right	SR, SHR
Shift Right with Link	SRL
Rotate Right	RR, ROR
Rotate Right with Link	RRL

The source statement format, the instruction format, and the description of the operation of each Shift, Rotate, or Serial Input/Output Instruction follow.

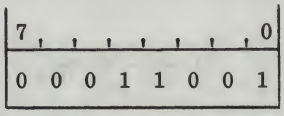
SERIAL INPUT/OUTPUT (SIO)

SOURCE STATEMENT

Mnemonic

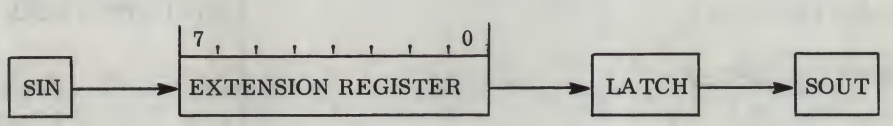
SIO

INSTRUCTION FORMAT



Operation: $(E_i) \rightarrow (E_{i-1})$, SIN $\rightarrow (E_7)$, $(E_0) \rightarrow$ SOUT

The contents of the Extension Register, (E), are shifted right one bit. The initial content of bit 0 is shifted to the serial data output latch and appears on the output pin SOUT. The signal on the data input pin SIN is shifted into bit 7.



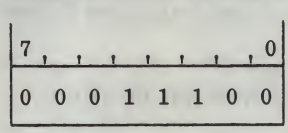
SHIFT RIGHT (SR, SHR)

SOURCE STATEMENT

Mnemonic

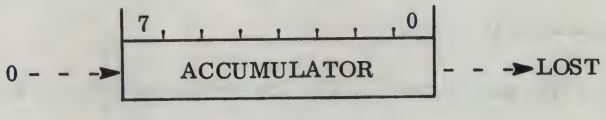
SR
SHR

INSTRUCTION FORMAT



Operation: $(AC_i) \rightarrow (AC_{i-1})$, $0 \rightarrow (AC_7)$

The contents of the Accumulator, (AC), are shifted right one bit. The initial content of bit 0 is lost. Zero is shifted into bit 7.



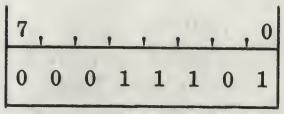
SHIFT RIGHT WITH LINK (SRL)

SOURCE STATEMENT

Mnemonic

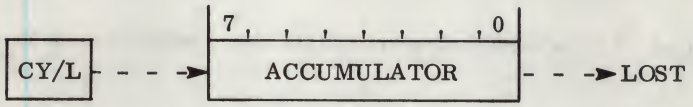
SRL

INSTRUCTION FORMAT



Operation: $(AC_i) \rightarrow (AC_{i-1})$, $CY/L \rightarrow (AC_7)$

The contents of the Accumulator, (AC), are shifted right one bit. The initial content of bit 0 is lost. The Link Flag, CY/L, is shifted into bit 7. The Link Flag is not altered.



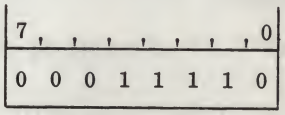
ROTATE RIGHT (RR, ROR)

SOURCE STATEMENT

Mnemonic

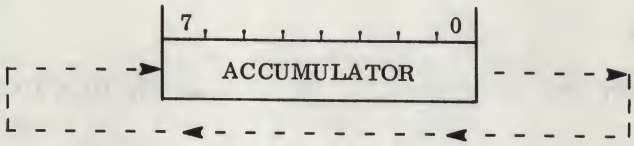
RR
ROR

INSTRUCTION FORMAT



Operation: $(AC_i) \rightarrow (AC_{i-1})$, $(AC_0) \rightarrow (AC_7)$

The contents of the Accumulator, (AC), are rotated right one bit. The initial content of bit 0 is shifted into bit 7.



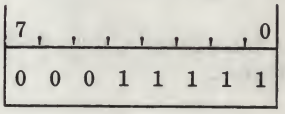
ROTATE RIGHT WITH LINK (RRL)

SOURCE STATEMENT

Mnemonic

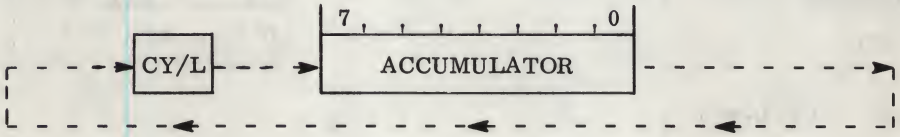
RRL

INSTRUCTION FORMAT



Operation: $(AC_i) \rightarrow (AC_{i-1})$, $(AC_0) \rightarrow CY/L \rightarrow (AC_7)$

The contents of the Accumulator, (AC), are rotated right one bit. The initial content of bit 0 is shifted into the Link Flag, CY/L, and the initial content of the Link Flag is shifted into bit 7 of AC.



4.1.8 Miscellaneous Instructions

There are nine instructions in this group, eight single-byte, and one double-byte. The Miscellaneous Instructions and mnemonics are as follows:

Halt	HALT
Clear Carry/Link	CCL
Set Carry/Link	SCL
Disable Interrupt	DINT
Enable Interrupt	IEN
Copy Status to AC	CSA
Copy AC to Status	CAS
No Operation	NOP
Delay	DLY

The source statement format, the instruction format, and the description of each Miscellaneous Instruction follow.

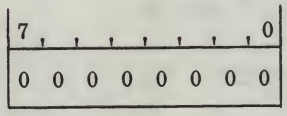
HALT (HALT)

SOURCE STATEMENT

Mnemonic

HALT

INSTRUCTION FORMAT



Operation: Pulse H-flag at I/O status time.

The H-flag and the CONTinue input may be combined to generate a programmed halt of the microprocessor. However, in a particular application system, this instruction may be used for functions other than HALT. For detailed information on the hardware operation of the halt instruction, see the SC/MP data sheet.

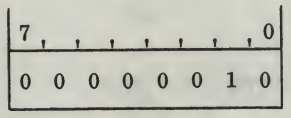
CLEAR CARRY/LINK (CCL)

SOURCE STATEMENT

Mnemonic

CCL

INSTRUCTION FORMAT



Operation: CY/L ← 0

The Carry/Link Flag, CY/L, in the Status Register is cleared. The remaining bits in the Status Register are not affected.

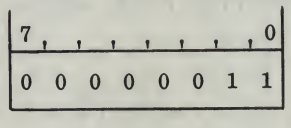
SET CARRY/LINK (SCL)

SOURCE STATEMENT

Mnemonic

SCL

INSTRUCTION FORMAT



Operation: CY/L ← 1

The Carry/Link Flag, CY/L, in the Status Register is set. The remaining bits in the Status Register are not affected.

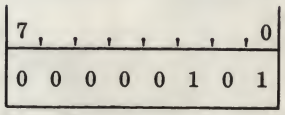
ENABLE INTERRUPT (IEN)

SOURCE STATEMENT

Mnemonic

IEN

INSTRUCTION FORMAT



Operation: IE ← 1

The Interrupt Enable Flag, IE, in the Status Register is set; the remaining bits in the Status Register are not affected. The processor interrupt system is enabled. Interrupts will be processed as received after the next instruction is fetched and executed. (See 2.5 and 5.6.2.)

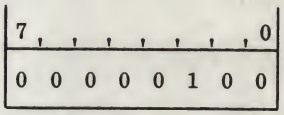
DISABLE INTERRUPT (DINT)

SOURCE STATEMENT

Mnemonic

DINT

INSTRUCTION FORMAT



Operation: IE ← 0

The Interrupt Enable Flag, IE, in the Status Register is cleared; the other bits in the Status Register are not affected. The processor interrupt system is disabled. Interrupts which occur while the system is disabled will not be processed.

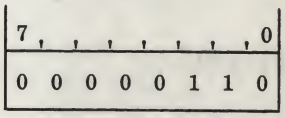
COPY STATUS TO AC (CSA)

SOURCE STATEMENT

Mnemonic

CSA

INSTRUCTION FORMAT



Operation: (AC) ← (SR)

The contents of the Status Register, (SR), replace the contents of the Accumulator, (AC). The initial contents of AC are lost; the contents of SR are not altered.

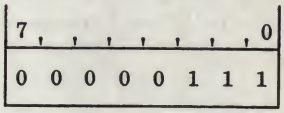
COPY AC TO STATUS (CAS)

SOURCE STATEMENT

Mnemonic

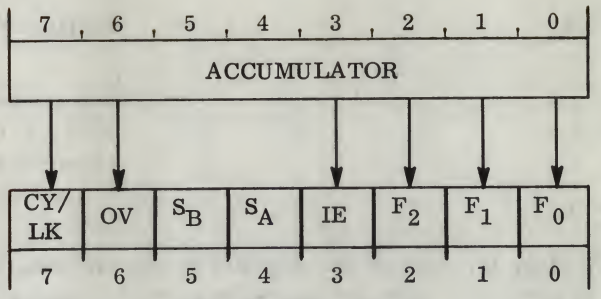
CAS

INSTRUCTION FORMAT



Operation: (SR) ← (AC)

The contents of the Accumulator, (AC), replace the contents of the Status Register, (SR). SR bits 4 and 5 are external sense bits and are not affected by this instruction. The initial contents of SR (except for bits 4 and 5) are lost; the contents of AC are not altered. See Status Register diagram on the following page.



STATUS REGISTER

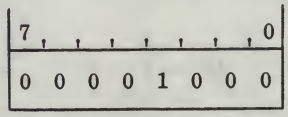
If IE is changed from 0 to 1 by this instruction, the interrupt system will be armed after the next instruction is fetched and executed (see 2.5 and 5.6.2).

NO OPERATION (NOP)

SOURCE STATEMENT

Mnemonic
NOP

INSTRUCTION FORMAT



Operation: (PC) ← (PC) + 1

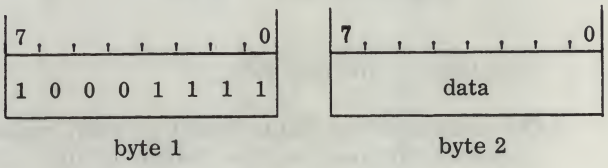
The contents of the Program Counter, (PC), are incremented by 1. The NOP instruction takes the minimum 5-microcycle execution time. Undefined opcodes encountered are considered to be one-byte or two-byte NOPs and may take 5 to 10 microcycles to execute depending on the code. It is recommended that only the Opcode 08 be used to insure software compatibility with future products.

DELAY (DLY)

SOURCE STATEMENT

Mnemonic	Operand
DLY	data

INSTRUCTION FORMAT



Operation: DELAY = 13 + 2(AC) + 2 data + 2^9 data

This instruction delays processing a variable length of time. The contents of the Accumulator, (AC), and the data byte are considered unsigned binary numbers (maximum value of each is 255). The number computed from the given equation is the execution time in microcycles. The following table gives some typical execution times. Range of delay is from 13 to 131593 microcycles.

AC

data	0	25	50	75	100	125	150	175	200	225
0	13	63	113	163	213	263	313	363	413	463
1	527	577	627	677	727	777	827	877	927	977
2	1041	1091	1141	1191	1241	1291	1341	1391	1441	1491
3	1555	1605	1655	1705	1755	1805	1855	1905	1955	2005
4	2069	2119	2169	2219	2269	2319	2369	2419	2469	2519
5	2583	2633	2683	2733	2783	2833	2883	2933	2983	3033
6	3097	3147	3197	3247	3297	3347	3397	3447	3497	3547
7	3611	3661	3711	3761	3811	3861	3911	3961	4011	4061
8	4125	4175	4225	4275	4325	4375	4425	4475	4525	4575
9	4639	4689	4739	4789	4839	4889	4939	4989	5039	5089
10	5153	5203	5253	5303	5353	5403	5453	5503	5553	5603

To determine AC and data for a specific number of microcycles (m) use the following equations:

$$\text{data} = \text{truncate } ((m-13)/514)$$
$$AC = ((m-13) - 514(\text{data}))/2$$

Using these equations, the delay time will be either exact or one microcycle less than the required number of microcycles.

4.2 COMMENT STATEMENTS

Comment statements are defined by a semicolon (;) in the first character position of the record. They do not generate code, but serve only to document the symbolic output listing of the program. For example:

```
;
;THIS IS A COMMENT STATEMENT
;
```

4.3 PSEUDO INSTRUCTION

A pseudo instruction is an assembler-dependent source statement that generates one or multiple machine-language instructions. Functionally, the pseudo instruction operates the same as a macro instruction (see chapter 6 for details on Macros). The difference between the two types of instructions is that the code to be generated by the pseudo instruction is defined in the assembler, while the code to be generated by the macro instructions is defined in the macro definitions, which are independent of the assembler (macro definitions normally reside in the user's program).

The Jump to Subroutine (JS) is the only pseudo instruction statement implemented in the SC/MP assembler. The statement is described on the following page.

JUMP TO SUBROUTINE (JS)

SOURCE STATEMENT

Mnemonic	Operands
JS	ptr, expression

Generated Code:

```
LDI     H(expression - 1)
XPAH    ptr
LDI     L(expression - 1)
XPAL    ptr
XPPC    ptr
```

H(expression - 1) and L(expression - 1) are not legal notation for the assembler, the assembler accepts only symbols in the H( ) and L( ) functions, and not expressions (symbol - 1 is considered an expression). What actually happens is that the assembler generates the value expression - 1 (taking into account page boundaries), and then applies the H( ) and L( ) functions to the value.

When a Jump to Subroutine is invoked, the code generated results in the setting of the specified Pointer Register, ptr, to the value, expression - 1. When calculating expression - 1, the 4K memory page structure of SC/MP is taken into account. As a result, the next instruction to be executed will always be the instruction addressed by expression.

This feature is important when addressing page boundaries. For example, if you wanted to jump to the beginning of page 3 (location X'2000), you might code the following sequence of instructions.

```
PG3     = PAGE3 - 1             ;SET PG3 = X'1FFF
        LDI     H(PG3)          ;LOAD X'1F INTO AC
        XPAH    P3              ;TRANSFER AC TO BITS 15-8 OF P3
        LDI     L(PG3)          ;LOAD X'FF INTO AC
        XPAL    P3              ;TRANSFER AC TO BITS 7-8 OF P3
        XPPC    3               ;PC NOW CONTAINS X'1FFF
          .
          .
          .
PAGE3:  .=X'2000
```

The above code would work for any location other than a page boundary. But, because of the wrap around feature of SC/MP (see 2.3.1, Paging Considerations), the next instruction fetched will be from location X'1000 rather than location X'2000. What you should do in this example is to code the following sequence of instructions.

```
        JS      P3, PAGE3
          .
          .
          .
PAGE3:  .=X'2000
```

When the JS instruction evaluates expression - 1 for a value to be used in the H( ) and L( ) functions, the assembler recognizes that expression (PAGE3) is a page boundary, so instead of setting value to X'1FFF it sets it to X'2FFF. Wrap around still occurs, but since the PC contains X'2FFF after the XPPC 3 instruction, the next instruction is fetched from location X'2000.

4.4 ASSIGNMENT STATEMENT

```
[label]        symbol =        expression        [;comments]
```

The Assignment Statement assigns the value of the expression on the right of the equals sign to the symbol on the left of the equals sign. The Statement may be preceded by a series of labels.

Example:

```
RETURN = 0D                    ;SYMBOL HAS CARRIAGE RET CODE
```

The Assignment Statement may set the location counter or refer to the current value of the location counter in an expression. The location counter assigns addresses to program statements at assembly time. It is similar to the Program Counter, which contains the address of the next instruction to be executed at execution time. As each instruction or data area is assembled, the location counter always points to the next available location in memory.

The period '.' is a special symbol used to specify the location counter. The location-counter symbol may appear on either side of the equals sign. If it appears on the left, it is assigned the value on the right side of the equals sign. The programmer may refer to the current setting of the location counter by referencing the '.' in the expression to the right of the equals sign. Assignment statements using the location-counter symbol are coded as free-form statements. For example:

```
        .=20                   ;SET LOCATION COUNTER TO 20
TABLE:  .=.+10                 ;RESERVE 10 LOCATIONS FOR TABLE
```

If the '.' appears on the left, the expression on the right must be defined during the first pass so subsequent label assignments may be made.

If the symbol on the left is not '.', then the expression on the right need not have a value during the first pass, but the expression must have a value during the second pass. This permits only one level of forward referencing. An example of more than one level of forward referencing follows:

```
FST:    A=B+2          This expression undefined during pass 2.
SND:    B=C-1          This expression undefined during pass 1.
THD:    C=25           This expression absolute
```

4.5 DIRECTIVE STATEMENTS

The Directive Statements control the assembly process and may generate data in the object program. The directive operator may be preceded by one or more labels, and may be followed by a comment. It occupies the operator field and is followed by either no operand or as many operands as are required by the particular operator.

Assembler directive operators and their functions are summarized in table 4-4. Note that all directive operators begin with a period for easy visual differentiation from the instruction operator mnemonics in the output listing. Each directive operator is described in more detail in the following paragraphs.

4.5.1 .TITLE Directive

```
[label]        .TITLE        symbol[,string]        [;comments]
```

The .TITLE directive identifies the load module in which it appears with a symbolic name and an optional definitive title. If a .TITLE directive does not appear in the program, the load module is given the name MAINPR. If more than one .TITLE directive is used, the last one encountered specifies the symbolic name.

The symbolic name and the string must meet the symbol and string construction restrictions discussed in chapter 3.

Example:

```
.TITLE  TBLKP,'TABLE LOOKUP'
```

Table 4-4. Summary of Assembler Directives

Directive	Function
.TITLE	Identification of program.
.END	Physical end of source program.
.LIST	Listing output control.
.SPACE	Space n lines in output listing.
.PAGE	Output Listing to top-of-form.
.BYTE	8-bit (single-byte) data generation.
.DBYTE	16-bit (double-byte) data generation.
.ASCII	Data generation for character strings.
.LOCAL	Establish a new local symbol region.
.IF	Conditional Assembly
.ELSE	
.ENDIF	
.FORM	Field Specification. ***
.ADDR	Address constant generation.
.SET	Assign values to variables.
.MACRO	Begin macro definition. *
.ENDM	End macro definition. *
.MLOC	Macro Local Symbols. *
.DO	Begin Macro-time looping. *
.ENDDO	End Macro-time looping. *
.EXIT	Exit Do loop. *
.IFC	Conditional Assembly. **
.ERROR	Macro error message generation. **
.MDEL	Macro delete. **

* Used only in macro definitions; see chapter 6 for descriptions.
** Macro related directive; see chapter 6 for descriptions.
*** Not in assemblers with macros.

4.5.2 .END Directive

[label] .END [address] [;comments]

The .END directive signifies the physical end of the source program. The optional address in the operand field may be either a symbol or a constant and indicates an execution address to a loader. In other words, it causes a branch to the address of the first executable instruction (entry point in contrast to load point) after the load is complete. The .END directive just makes the entry point address available to a loader. Use of this feature is dependent on the loader program.

Examples:

1. No entry point specified

```
LAST:    .END
```

2. Jump to the entry point at X'00A9

```
         .END  X'00A9
```

3. Jump to the entry point labeled START

```
         .END  START
```

Loader Dependent Option

4.5.3 .LIST Directive

[label] .LIST immediate [;comments]

The .LIST directive controls listing of the source program. This includes listing in general, listing of unassembled code caused by the .IF and .IFC directives, listing of macro expansions, and listing of generated code.

Control of the various list options depends upon the state of the five least significant bits of the evaluated expression in the operand field. Table 4-5 shows the options available in the order of their priority.

Options are usually combined to give the desired type of listing. Some examples follow:

1. Full listing:

```
.LIST 01
```

2. Full listing and list all code expanded during macro calls.

```
.LIST 0D
   or          <-- Full list
.LIST 01 ! 0C  <-- list all code expanded during macro calls
```

3. Suppress listing.

```
.LIST 0
```

Table 4-5. List Options

Function	Bit	Value	Description
Master List Control	0	1 0	* Full listing Suppress all listing
.IF List Control	1	1 0	Full listing (of .IF's and .IFC's) * Suppress unassembled code
Macro List Control	2,3	11 10 00	List all code expanded during macro calls List only code generated by macro calls * List only macro calls
Binary List Control	4	1 0	* List all the binary output by statements generating more than one word (e.g., .ASCII) List only the first two bytes of generated data

* indicates default

4.5.4 .SPACE Directive

[label] .SPACE immediate [;comments]

The .SPACE directive skips forward a specified number of lines on the output listing.

Examples:

1. Skip 20 (decimal) lines

```
.SPACE 20
```

2. Skip 20 (hexadecimal) lines

```
.SPACE 020
   or
.SPACE X'20
```

4.5.5 . PAGE Directive

[label] . PAGE [string] [;comments]

The . PAGE directive spaces forward to the top of the next page on the output listing. The optional string is printed as a page title on each page until a . PAGE directive containing a new string is encountered. No action is taken (except for a new page title) if the . PAGE directive is encountered immediately after an assembler generated top-of-page request.

Example:

```
.PAGE   'TTY I/O ROUTINES'
```

4.5.6 . BYTE Directive

[label] . BYTE expression[,expression...] [;comments]

The . BYTE directive stores consecutively in memory one 8-bit byte for each given expression. If the directive has a label, it refers to the address of the first expression. The value of each expression must be in the range, -128 to +127 for signed data or 0 to 255 for unsigned data.

Examples:

1. Single expression without a label

```
.BYTE   X'FF
```

2. Multiple expressions with a label

```
TBL:  .BYTE  MPR+10,X'FF,X'00
```

NOTE

TBL is assigned the location counter value of the byte containing the expression MPR+10.

4.5.7 . DBYTE Directive

[label] . DBYTE expression[,expression...] [;comments]

The . DBYTE directive stores 16-bit data in two consecutive 8-bit memory locations. Each expression of a . DBYTE directive is evaluated, and its value is placed in the next available pair of memory locations. The value of each expression must be in the range, -32768 to +32767 for signed data or 0 to 65535 for unsigned data.

The . DBYTE directive generates 16-bit address constants for use with Memory Reference or Memory-Increment/Decrement Instructions (4.2.1 and 4.2.2). If the directive has a label, it refers to the memory address of the first byte generated by the directive.

Examples:

1. Without a label

```
.DBYTE   X'77FF
```

2. With a label

```
LABL:  .DBYTE   X'77FF
```

NOTE

LABL is assigned the value of the location of X'77 and LABL + 1 is assigned the value of the location of X'FF.

A single character string appearing in a . DBYTE generates an 8-bit right-justified constant.

4.5.8 .ADDR Directive

[label] .ADDR expression[,expression...] [;comments]

The .ADDR directive generates 16-bit address constants to be used by transfer instructions. Each expression in the directive is evaluated, is decremented by 1, and then is placed in the next available pair of memory bytes. The decrementing takes into account the modulo-4096 address arithmetic used in SC/MP.

The effect of this directive is that if a pointer register is loaded with the resulting constant and its contents are exchanged with the program counter, the next instruction to be executed will be the one addressed by the value, expression.

Example:

```
AD1:    .ADDR    OUTPUT
          .
          .
          .
        LD       AD1
        XPAH     P3
        LD       AD1+1
        XPAL     P3
        XPPC     P3
```

The subroutine OUTPUT is executed.

4.5.9 .ASCII Directive

[label] .ASCII string[,string...] [;comments]

The .ASCII directive stores data in successive memory locations by translating the characters in the string into their 7-bit ASCII equivalent code. Each string must be enclosed in single quote marks ('). Each character occupies one byte in memory. The .ASCII directive is used primarily to generate messages for output on teletypewriter or printer.

Example:

```
        .ASCII    'INPUT OF VALUE OF X'
```

4.5.10 .LOCAL Directive

[label] .LOCAL [;comments]

The .LOCAL directive establishes a new program section for local symbols (symbols beginning with a dollar sign ($)). Designated symbols between two .LOCAL directive statements have the value assigned to them only within that particular section of the program. Note that a .LOCAL directive is assumed at the beginning and the end of a program; thus, one .LOCAL directive within a program divides the program into two sections.

If the first character of a symbol is a dollar sign ($), the assembler attaches a unique character from the ASCII character set to the end of the symbol. Initially, this character is an exclamation point '!' (X'21). Each time a .LOCAL directive is encountered, the value of the added character is advanced by one with the letter "Z" (X'5A) as the last legal value. Therefore, up to 58 .LOCAL directives can appear in one assembly.

Example:

```
        .LOCAL
```

4.5.11 Conditional Assembly Directives

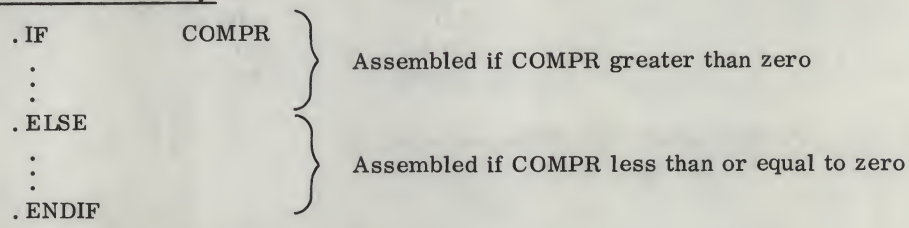
[label]	.IF	expression	[;comments]
	.ELSE		[;comments]
	.ENDIF		[;comments]

The conditional assembly directives selectively assemble portions of a source program based on the value of the expression in the .IF directive statement.

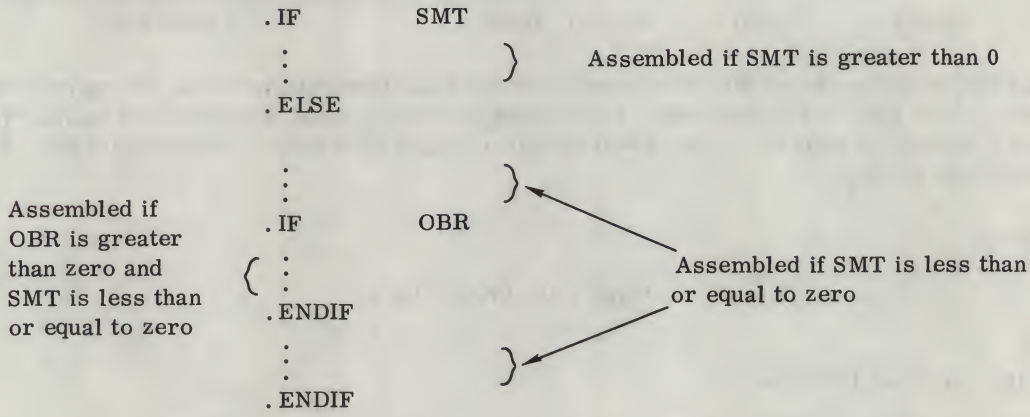
All source statements between a .IF directive and its associated .ENDIF are defined as a .IF-.ENDIF block. These blocks can be nested to a depth of ten. The .ELSE directive can be included optionally in a .IF-.ENDIF block. The .ELSE directive segments the block into two parts. The first part of the source statement block is assembled if the .IF expression is greater than zero; otherwise, the second part is assembled. When the .ELSE directive is not included in a block, the block is assembled only if the .IF expression is greater than zero. If an error is detected in the expression, the assembler assumes a true value (greater than zero).

Examples:

1. Two-part conditional assembly



2. Nested .IF-.ENDIF block conditional assembly



Labels appearing on .IF statements are assigned the address of the next assembled instruction. Labels cannot be used on .ELSE or .ENDIF statements.

Listing of unassembled code may be controlled by appropriate use of .LIST directives (see paragraph 4.5.3).

4.5.12 .FORM Directive *

[label]	.FORM	symbol, exp[(exp)] [, exp(exp)...]	[;comments]

The .FORM directive specifies a field format for an 8-bit byte and optionally presets bits of the word to an initial value. The .FORM directive may be used for generating special instructions not recognized by the assembler.

* Not available in the SC/MP (PACE) Cross Assembler

Symbol is the name of the byte format. Format is invoked by placing the symbol in the operation field of a statement. Note that symbols assigned values by the .FORM directive can be used only as operation symbols.

The expressions that follow the symbol specify field lengths within the byte and, optionally, the expressions contained in parentheses specify values for the specified fields starting at the most significant bit. If a value is specified for a field, it must be enclosed in parentheses and immediately follow the length attribute.

Examples:

1. .FORM CK,2,4,2(X'3) divides the CK into 3 fields: one 2-bit field, one 4-bit field, and one 2-bit field with the preset value X'3. The preset value field cannot be changed when the format is invoked.

2. Generate binary strings

```
.FORM    BINARY,1,1,1,1,1,1,1,1
```

When the BINARY format is invoked by

```
BINARY   1,0,1,1,0,0,1,1
```

It causes the number $B3_{16}$ to be generated.

4.5.13 .SET Directive

```
[label]          .SET          symbol,expression          [;comments]
```

The .SET directive is used to assign values to reassignable (set) variables. A variable assigned a value with the .SET directive can be reassigned different values an arbitrary number of times.

Examples:

```
.SET     A,100              ;SET A = 100
.SET     B,50               ;SET B = 50
.SET     C,A-25*B/4         ;SET C = A - 25 * B/4
```

NOTE

This expression is always evaluated from left to right regardless of the operators used between the variables and constants.

Chapter 5

PROGRAMMING TECHNIQUES

This chapter discusses the programming techniques used to produce efficient SC/MP object code. Examples of coding are included to illustrate the method by which the techniques are implemented.

5.1 STACK PROGRAMMING

A convenient way of temporarily saving status and return addresses from subroutines and interrupt service routines is to maintain a stack in read/write memory. An advantage of a software stack compared to a hardware stack is that the hardware stack is limited in size to a fixed number of storage locations; any additional data pushed onto a stack cause an overflow and loss of data at the bottom of the stack. A software stack, on the other hand, virtually can be made any length, so overflow cannot occur. Another advantage of using software stacks is that more than one stack can be maintained.

System software may use the following pointer register assignments:

Pointer Register	Function
P1	ROM Pointer and miscellaneous
P2	Stack Pointer
P3	Subroutine Pointer

Storing and retrieving data from the stack are accomplished by the following methods:

1. Store one byte of data or address

```
CE00        ST      @-1(P2)        ;PUSH A BYTE ONTO THE STACK
```

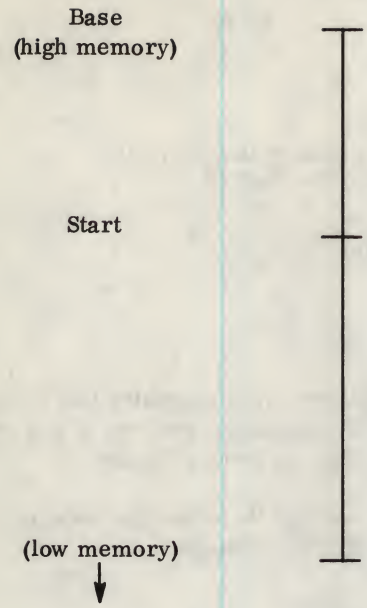
2. Retrieve one byte of data or address

```
C601        LD      @1(P2)         ;PULL A BYTE OFF THE STACK
```

It should be noted that the auto-indexing feature is used to move the stack pointer address up or down the stack. The stack pointer (P2) always points to the last value pushed onto the stack.

5.1.1 Stack Operations

Using the stack conventions previously stated creates a stack that begins in high memory and extends downward. The most effective method of using this stack consists of fixing the base location of the stack, allocating any permanent locations required by the program, and then allowing the dynamic portion of the stack to expand and contract below that. For example see the following page.



Permanent Area — contains global data for one or more programs. Fixed structure.

Dynamic Area — used by a particular program or subroutine at a particular time. The structure of this area depends upon the state of the system.

The permanent area of the stack always may be accessed by providing two words (labeled STKPT) for saving the stack pointer and then using the following code.

```
                .PAGE
0004            BASE    =.              ;PERM. AREA OF STACK
                 .       .
                 .       .
                 .       .
                        .=.-2
FFFE            STKPT   =.-BASE
                 .       .
                 .       .
C404            LDI     L(BASE)
32              XPAL    P2
01              XAE                     ;SAVE LO-HALF OF PTR
C400            LDI     H(BASE)
36              XPAH    P2
CAFE            ST      STKPT(P2)
40              LDE
CAFF            ST      STKPT+1(P2)
```

Loading a Pointer from a word-pair pointed to by the same pointer.

```
C200            LD      0(P2)           ;LOAD UPPER POINTER ADDRESS
01              XAE                     ;SAVE
C201            LD      1(P2)           ;LOAD LOWER POINTER ADDRESS
32              XPAL    P2              ;TRANSFER TO LOWER P2
40              LDE                     ;RESTORE UPPER ADDRESS
36              XPAH    P2              ;TRANSFER TO UPPER P2
```

5.1.2 Repeatable Subroutine Calls

If P3 is being used as a subroutine pointer, the subroutine may be called repeatably without reloading P3 as long as P3 is not disturbed. The subroutine must be set up as follows:

```
          SIN:
           :          :
           :          :
           :          :
3F                XPPC    P3              ;SUBROUTINE RETURN
90FD              JMP     SIN             ;FOR REENTRY
```

All other things being equal, subroutines should be coded as repeatable.

5.2 SUBROUTINES

Because of the problems involved when a program crosses a page boundary, it is suggested that the programmer code his programs in modules smaller than a page. Organizing code into small subroutines is a more efficient way of coding, since it is easier to verify several small subroutines than one large program.

There are two methods used to implement subroutines, depending on whether the subroutine is single level or nested. Nesting is a condition where a subroutine contains calls to other subroutines.

5.2.1 Multilevel Subroutines

To implement multilevel (nested) subroutines, a stack must be created in memory. As an example, pointer P2 could be defined as the stack pointer. The address loaded into P2 would point to the top of the stack. This address would be a location in read/write memory.

If nesting is not required, subroutines can be called using the pointer registers to save the return address.

The following examples assume that SUBR is the label on the first instruction of the subroutine. It should be noted that the examples will not work on a page boundary (see 4.3, Pseudo Instruction, for the explanation).

SUBROUTINE JUMP

```
0015  SUBR1    =       SUBR-1
C415            LDI     L(SUBR1)        ;LOAD LOWER SUBROUTINE ADDRESS
33              XPAL    P3              ;TRANSFER LOWER TO P3L
C400            LDI     H(SUBR1)        ;LOAD UPPER SUBROUTINE ADDRESS
37              XPAH    P3              ;TRANSFER UPPER TO P3H
3F              XPPC.   P3              ;EXCHANGE PC AND P3
```

SUBROUTINE RETURN

```
3F              XPPC    P3              ;RETURN FROM SUBROUTINE EXCHANGE
```

If multilevel subroutines are used, the current contents of the pointer register should be saved on the top of the stack and should be restored upon return from the subroutine.

```
C415            LDI     L(SUBR1)        ;LOAD LOWER SUBROUTINE ADDRESS
33              XPAL    P3              ;TRANSFER TO P3
CEFF            ST      @-1(P2)         ;SAVE P3L ON STACK
C400            LDI     H(SUBR1)        ;LOAD UPPER SUBROUTINE ADDRESS
37              XPAH    P3              ;TRANSFER TO P3H
CEFF            ST      @-1(P2)         ;SAVE P3H ON STACK
3F              XPPC    P3              ;JUMP TO SUBROUTINE
C6FF            LD      @-1(P2)         ;RETURN FROM SUBROUTINE, LOAD P3H
                                        ;FROM STACK
37              XPAH    P3              ;TRANSFER TO P3H
C601            LD      @1(P2)          ;LOAD P3L FROM STACK
33              XPAL    P3              ;TRANSFER P3L, BACK TO NORMAL
```

5.2.2 Jump Immediate

A jump immediate can be implemented directly using the subroutine jump as shown in example 5.2.1 but without executing a subroutine return. This facility allows a jump to any address in memory.

5.2.3 Conditional Subroutine Jumps

Conditional subroutine jumps can be implemented using a condition jump test to bypass the subroutine call. For example:

```
9400          JP      NOJSR           ;NO SUBROUTINE JUMP IF AC POSITIVE
                :       :
                :       :
                :       SUBROUTINE CALL AS IN OTHER EXAMPLES
                :       :
          NOJSR:
                :       :
                :       :
```

5.2.4 Multiple Subroutine Return

The same programming technique described in 5.2.1 can be used to establish more than one return address after a subroutine has been executed. This technique can be used to test a flag condition and to branch conditionally to one of two or more locations, depending upon the condition of the flag. For example, a subtract routine might require three returns: one for a positive result, one for a negative result, and one for a zero result.

The example below affects a conditional call from a subroutine on the current page.

```
C432          LDI     LOWER           ;LOAD LOWER RETURN ADDRESS
33            XPAL    P3
CEFF          ST      @-1(P2)         ;PUSH ONTO STACK
C433          LDI     UPPER           ;LOAD UPPER RETURN ADDRESS
37            XPAH    P3
CEFF          ST      @-1(P2)         ;PUSH ONTO STACK
3F            XPPC    P3              ;JUMP TO SUBROUTINE
90EE          JMP     ZERO            ;ZERO RETURN
90EC          JMP     POS             ;POSITIVE RETURN
C8ED          ST      RESULT          ;NEGATIVE RETURN
                :       :
          TEST1:
                :       :
                :       :
C601          LD      @1(P2)          ;LOAD UPPER RETURN ADDRESS
37            XPAH    P3              ;TRANSFER TO P3 HIGH
C601          LD      @1(P2)          ;LOAD LOWER RETURN ADDRESS
33            XPAL    P3              ;TRANSFER TO P3 LOW
40            LDE                     ;GET RESULTS FROM EXTENSION
9807          JZ      RETURN          ;JUMP IF AC=0
C702          LD      @2(P3)          ;INCREMENT RETURN ADDRESS
40            LDE                     ;GET RESULTS FROM EXTENSION
9402          JP      RETURN          ;JUMP IF AC>0
C702          LD      @2(P3)          ;AC<0, INCREMENT RETURN
3F      RETURN: XPPC  P3              ;EXCHANGE P3 AND PC
```

5.2.5 Transferring Data to Subroutines

Frequently parameters must be passed to a subroutine when it is called; this is accomplished by listing the parameters in the bytes following the subroutine call and by incrementing the return address to the next executable instruction. Below is an example of the coding for this data-transfer technique:

```
C400            JS      P3,MATH         ;JUMP TO SUBROUTINE
37C4
6433
3F
01              .BYTE   X'01            ;2 BYTES PASSED TO
02              .BYTE   X'02            ;SUBROUTINE
                 .       .
                 .       .
                 .       .
C701  MATH:     LD      @1(P3)          ;ADJUST PTR TO 1ST PARAMETER
C701            LD      @1(P3)          ;FETCH PARAMETER
C8EF            ST      PARM1
C300            LD      (P3)            ;FETCH PARAMETER 2
C8EC            ST      PARM2
```

At this point, the return address is in P3. The programmer may elect to leave it in P3, save it on the stack or store it locally until it is needed to return from the subroutine.

It also may be convenient to store all of the subroutine input parameters on the stack before calling the subroutine and then to have the subroutine place any output parameters on the stack before executing a return.

5.3 LOOP COUNTER

When executing a routine in which a group of instructions is repeated a given number of times, it may be convenient to use a memory location as a counter register. The address of the memory location used as a counter would be stored in one of the pointer registers.

An advantage of the use of a memory location as a counter rather than as an internal register (such as E) that is the Increment and Load (ILD) and the Decrement and Load (DLD) Instructions associated with memory-location increments and decrements do not affect the value of the carry bit in the status register. This is particularly important in serial arithmetic operations, where the carry bit must be saved for the next step.

It should be noted that both the ILD and DLD instructions destroy the contents of the accumulator, so the contents of the accumulator should be saved temporarily if they are needed in additional calculations.

The following exemplifies the implementation of a memory counter for a program containing a loop that is to be executed eight times.

```
C400            LDI     H(CNTR)         ;LOAD HIGH ORDER ADDRESS OF COUNTER
35              XPAH    P1              ;P1 AS COUNTER POINTER
C41E            LDI     L(CNTR)         ;LOAD LOW ORDER ADDRESS OF COUNTER
31              XPAL    P1              ;P1 AS COUNTER POINTER
C408            LDI     8               ;LOAD NUMBER OF TIMES TO LOOP
C900            ST      (P1)            ;STORE COUNTER VALUE IN MEMORY
                 .       .
                 .       .
                 .       .
        LOOP:
                 .       .
                 .       .
                 .       .
33              XPAL    P3              ;SAVE AC IN P3L
B900            DLD     0(P1)           ;DECREMENT COUNTER
9803            JZ      NEXT            ;IF COUNTER=0, END OF LOOP
33              XPAL    P3              ;RECOVER AC FROM P3L
90F8            JMP     LOOP            ;REPEAT LOOP
33      NEXT:   XPAL    P3              ;RECOVER AC FROM P3L
```

In a similar manner, the counter and temporary storage for the AC can be saved on the stack, thereby eliminating the overhead of initializing P1 (in the preceding example). The Extension Register may also be used as temporary storage for saving the Accumulator.

5.4 PAGE CONSIDERATIONS

PC-relative memory-reference instructions can reference memory only within the current page (4096 bytes), and then only ±127; this requires the programmer to take certain precautions and to use the techniques described in this section to avoid problems at page boundaries.

5.4.1 Instructions at the Page Boundary

The program counter does not automatically increment across page boundaries, but does effect a "wrap-around" in the same page. Therefore, a 2-byte instruction might occupy the last two bytes of a page or the last and first byte of the same page, but not the last byte of one page and the first byte of the next page. The assembler flags the condition of a 2-byte instruction whose first byte is at the page boundary so the programmer can modify the source code.

5.4.2 Programs Residing Across Page Boundaries

Since PC-relative memory references are limited to the page occupied by the instruction, the simplest way of writing a program is to organize it as short subroutines, each of which resides within one page of memory. It is advisable usually not to fill a page with one subroutine, since corrections that require additional program steps could not easily be incorporated.

If the user wants to continue his program across a page boundary, he must explicitly set the location counter and he must use the Jump to Subroutine (JS) pseudo instruction. For example:

```
        JS    P3,PAGE3              ;JUMP TO PAGE 3
        .
        .
        .
PAGE3:  .=X'2000
```

This example jumps to first location of page 3. For any location other than the first location of a page, the code shown in 5.2.1 will work (see 4.3, Pseudo Instruction, for the explanation). The user should be aware that if he uses the JS instruction, the contents of the Accumulator are lost.

Using indexed addressing, it is relatively simple to write programs that occupy more than one page of memory. The first instruction on each page loads a pointer register with the alternate page address; indexed addressing is used to reference the alternate page, and PC-relative addressing is used to reference the current page. The techniques used to load a subroutine address apply here.

5.5 TEXT PROGRAMMING TECHNIQUES

When programs require extensive dialog, textual display, or printout, attention should be given to the technique that programs the textual printout, since it is likely to be subject to modification. One technique that readily lends itself to the SC/MP is the literal pool. A literal pool is an area within an assembled program where the literals used within the program are stored. Within the literal pool, all duplication of text is eliminated. For example, consider the following five messages:

1. ENTER 5 COEFFICIENTS
2. COEFFICIENT OUT OF ALLOWED RANGE. RE-ENTER
3. ANSWER =
4. ANOTHER?
5. NO VALID ANSWER. RE-ENTER 5 COEFFICIENTS

Text for the five messages may be stored in a literal pool as shown below.

```
2E52  L1:     .ASCII  '.RE-'
452D
454E  L2:     .ASCII  'ENTER '
5445
5220
35    L3:     .ASCII  '5'
2043  L4:     .ASCII  ' COEFFICIENT'
4F45
4646
4943
4945
4E54
53    L5:     .ASCII  'S'
204F  L6:     .ASCII  ' OUT OF ALLOWED RANGE'
5554
204F
4620
414C
4C4F
5745
4420
5241
4E47
45
414E  L7:     .ASCII  'ANOTHER?'
4F54
4845
523F
414E  L8:     .ASCII  'ANSWER'
5357
4552
3D20  L9:     .ASCII  '= '
4E4F  L10:    .ASCII  'NO VALID '
2056
414C
4944
20
      L11:    .=.+1
```

Messages are created by indexing the literal pool using a 2-byte repeating sequence. Byte 1 holds the displacement from the base of the literal pool to the first required character. Byte 2 holds the value of the number of characters to be printed. If the first byte of the byte pair holds the value X'FF, it signifies the end of the message; otherwise, another segment of the message is sought in the next byte pair. Each of the five messages described above could be created by the index sequence shown below.

```
04    I1:     .BYTE   L2-L1           ;ENTER 5 COEFFICIENTS
14            .BYTE   I6-I2
FF            .BYTE   X'FF
08    I2:     .BYTE   I4-I1           ;COEFFICIENT
0C            .BYTE   L5-L4
18            .BYTE   L6-L1           ;CUT OF ALLOWED RANGE
15            .BYTE   L7-L6
00            .BYTE   L1-L1           ;.REENTER
0A            .BYTE   L3-L1
FF            .BYTE   X'FF
35    I3:     .BYTE   L8-L1           ;ANSWER=
08            .BYTE   I10-I8
FF            .BYTE   X'FF
2D    I4:     .BYTE   L7-L1           ;ANOTHER?
08            .BYTE   L8-L7
FF            .BYTE   X'FF
3D    I5:     .BYTE   L10-L1          ;NO VALID ANSWER
09            .BYTE   I11-I10
35            .BYTE   L8-L1
06            .BYTE   I9-I8
00            .BYTE   L1-L1           ;.REENTER 5 COEFFICIENTS
18            .BYTE   I6-I1
FF            .BYTE   X'FF
```

A subroutine generates the printed messages. To write a message, the procedure is to call the subroutine and to specify the index that identifies the message to be printed. For example, to print "Answer=", the call would be as shown below.

```
3F              XPPC    P3              ;WRITE "ANSWER="
00D0            .CBYTE  I3
                                        ;RETURN POINT
```

Subroutine WRIT calculates the section of the literal pool to be printed. The first character is at the address: L1 plus the contents of I3. The number of characters to be printed is derived from the contents of I3 + 1. The next byte contains X'FF, so printing is finished; otherwise, printing would continue with the next pair of index bytes specifying the next string of characters.

5.6 INPUT AND OUTPUT PROGRAMMING TECHNIQUES

The programming of data transfers between read/write memory and peripheral devices is generally classified as input/output programming. Depending on the significance of the input/output operations in the overall program, different approaches to input/output program implementation are recommended; these approaches are described in the following sections.

5.6.1 Programmed Input/Output

A programmed input/output operation is initiated and completed under the control of the initiating program. In figure 5-1, the program being executed starts the input/output operation; then, the program waits for the operation to be completed before continuing.

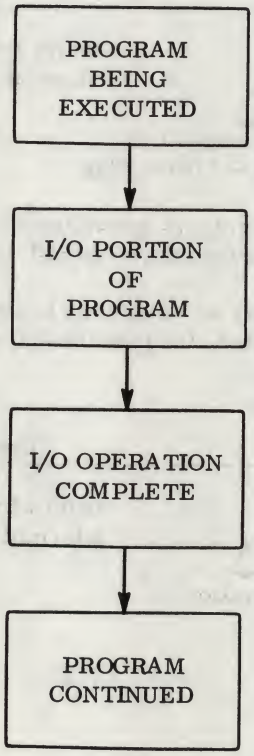


Figure 5-1. Programmed Input/Output

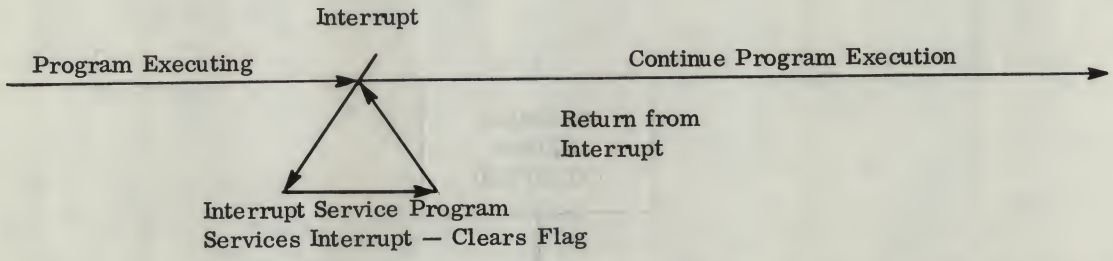
SC/MP allows any memory-reference instruction to execute a programmed input/output operation. Peripheral device controllers are assigned specific memory addresses, any of which, when referenced by a memory-reference instruction, executes the input/output operation. It is necessary that the memory addresses assigned to the peripheral device be unique to the device; that is, no other peripheral device uses the same assigned memory addresses, nor is there memory with the same addresses. Also, the device controller must contain the necessary logic to decode its assigned addresses, and, then, to gate data on and off the data bus. For example, memory addresses X'7FFF and below might be reserved for read/write or read-only memory and memory addresses X'8000 to X'FFFF might be reserved for peripheral device controllers. The user, however, can define his own convention, as described in the SC/MP Data Sheet.

The actual program steps required to enable programmed input/output depend on the design of the device controller.

5.6.2 Interrupt Input/Output

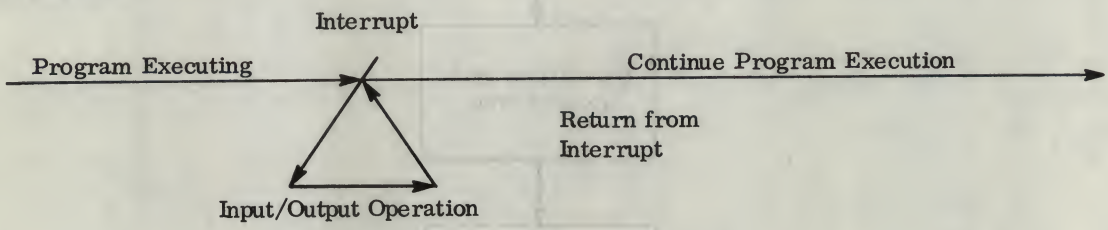
In certain cases, an input/output operation initiated by a program requires a significant length of time (many milliseconds) for execution; during this time, the program might perform other tasks. In other cases, the frequency of input/output service that requires the use of a certain input/output device might be such that it would be convenient for the program to ignore the device unless it specifically requires service. Each of these situations may be handled by taking advantage of the SC/MP interrupt system and by employing interrupt input/output for devices that have interrupt capability.

In figure 5-2, the program might initiate the input/output operation as part of its normal sequence-of-operation and might set a flag indicating that such action was taken. An input/output device that has interrupt capability, upon completion of an input/output operation, transmits an interrupt to SC/MP to indicate completion of the operation.



The flag may be employed by the original program to determine whether or not the input/output operation has been completed and whether or not the input/output device is still busy.

Another way that an input/output operation may be initiated is to transmit an interrupt signal to the CPU. In this case, the program being executed is interrupted, the input/output operation is effected, and, then, the interrupted program is resumed.



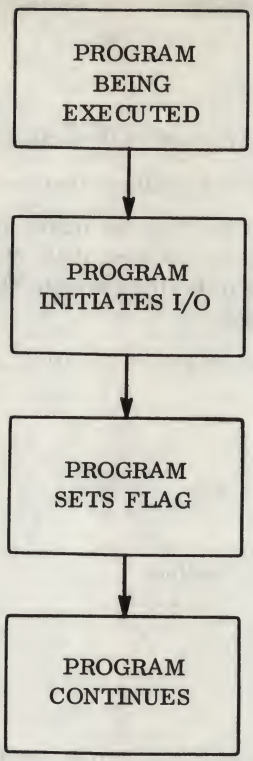


Figure 5-2. Interrupt Input/Output Initiation

Interrupt input/output requires a definite and specific sequence of events, irrespective of what peripheral device is to be serviced; the following sequence occurs.

1. SC/MP assumes that if an interrupt is to be processed, P3 contains a pointer to the interrupt service routine. Therefore, this pointer must be set to the proper address before interrupts are enabled.

2. In order for an interrupt to be accepted by SC/MP, the interrupt system must be enabled. The system is enabled by executing an instruction after setting the Interrupt Enable (IE) flag (bit 3) in the status register to a '1'. The IE flag is set to 1 by the IEN (Interrupt Enable) Instruction or by the CAS (Copy Accumulator to Status) Instruction with a '1' in bit 3 of the accumulator.

3. Once the interrupt system is enabled, a '1' in the Sense A input causes the following sequence of events:

   a. The instruction currently being executed is completed.
   b. Interrupts are disabled (IE cleared). Therefore, no further interrupts are accepted by SC/MP until interrupts are re-enabled.
   c. The contents of the PC are exchanged with the contents of P3.
   d. The instruction located at the location specified by the new (PC) + 1 is executed.

4. The instructions at entry to the interrupt service routine must perform a number of housekeeping tasks before the required input/output operation can proceed. Tasks, in order of normal execution, are as follows:

   a. Save the contents of registers that will be used by the interrupt routine so they can be restored just before returning from the interrupt. Register contents are saved typically on a stack, usually the same stack being maintained for subroutines.
   b. Determine the source of the interrupt. The way this is done depends on the design of the peripheral device controllers, but, usually, controllers are designed to respond to an interrupt acknowledge signal by transmitting a data byte (or word) that identifies the source of the interrupt. This acknowledge signal could be a particular address to which all interrupting devices respond.
   c. Once the interrupt has been identified, jump to the routine that services the identified device.

4. Execute input/output service routine of the selected device.

5. Restore the appropriate register contents that were saved in step 3a.

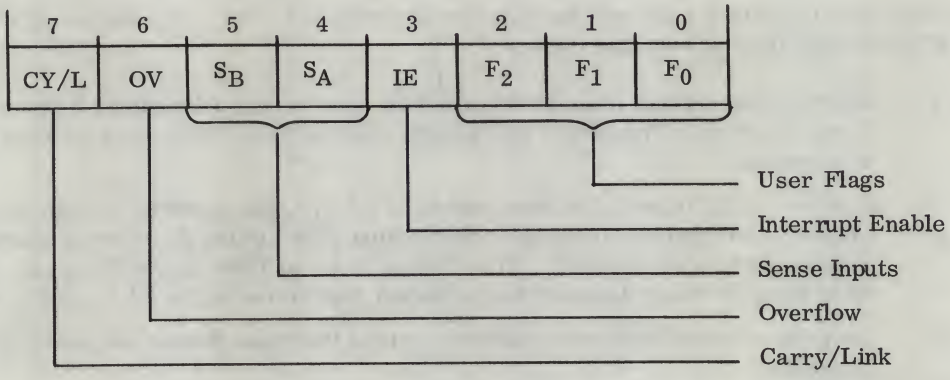
6. Return from the interrupt by enabling the interrupt, and then, by exchanging the return address in P3 (step 2c) with the program counter; thus, the execution continues at the program instruction following the interrupt. The following example shows the technique to insure the contents of P3 upon return from the interrupt.

7. The interrupt can be disabled by use of the DINT Instruction.

```
05      $RETURN: IEN                    ;ENABLE INTERRUPTS
3F               XPPC   P3              ;INTERRUPT SYSTEM IS ARMED AFTER
                                        ;THIS INSTRUCTION IS FETCHED.
        INTSVC:  •••                    ;INTERRUPT SERVICE STARTS HERE
                 <Service Routine>
9000             JMP    $RETURN
```

5.7 USING THE STATUS REGISTER

The status register is an 8-bit register, where 2 bits automatically reflect the result of accumulator operations; 1 bit is the interrupt enable, and the remaining 5 bits are sense bits or user flags. The Sense-A input also serves as the interrupt request input.



The status register bits are modified as indicated in table 5-1.

Table 5-1. Status Register Bits

Flag	Automatic Operation	Special Instruction	CAS Instruction
CY/L	Set/reset by arithmetic operations and rotate with link	SCL sets CCL resets	Loads from AC, bit 7
OV	Set/reset by arithmetic operations	——	Loads from AC, bit 6
S_B	Reflects input line	——	Not affected, read only
S_A	Reflects input line	——	Not affected, read only
IE	Reset by interrupt	IEN sets DINT resets	Loads from AC, bit 3
F_2	——	——	Loads from AC, bit 2
F_1	——	——	Loads from AC, bit 1
F_0	——	——	Loads from AC, bit 0

All bits of the Status Register except the sense inputs can be set or cleared by the CAS Instruction.

The CY/L and OV bits are set or cleared automatically according to the result of accumulator operations, as described below. CY/L also can be set by the SCL Instruction or can be cleared by the CCL Instruction. The IE bit is cleared automatically when an interrupt is accepted. It can be set by the IEN Instruction and can be cleared by the DINT Instruction. The two sense bits reflect the state applied to the external sense pins. The 3 general-purpose flag bits are set or cleared only as directed by the CAS Instruction.

The Status Register is a passive depository of status information, and apart from the operations described for CY/L, OV, and IE, all status operations (for example, testing of status or setting/clearing the overflow or flag bits) take place in the accumulator. Thus, the normal sequence of status-register operations is as follows:

1. Transfer status to accumulator.
2. Test/set/clear status.
3. Return accumulator contents to status register if update is required.

The contents of the Status Register may be tested by copying SR to AC, masking with a logical instruction, and testing with a conditional jump as shown in the test for overflow example below:

```
06          CSA                 ;COPY STATUS TO AC
D440        ANI     X'40        ;CLEAR ALL BITS EXCEPT 6 (OV)
9CFB        JNZ     OVFL        ;IF OVERFLOW, JUMP
```

Individual bits may be set or cleared using the following methods:

```
06          CSA                 ;COPY STATUS TO AC
D4FE        ANI     X'FE        ;SET UP AC TO CLEAR F0
DC02        ORI     2           ;SET UP AC TO SET F1
07          CAS                 ;CLEAR F0 AND SET F1
```

5.7.1 Arithmetic Operations

Arithmetic operations may be signed or unsigned. Consider first one byte. Unsigned, its numbering range is as follows:

$$\text{from } 0_{10} \text{ (X'00)} = 0\,0\,0\,0\,0\,0\,0\,0$$
$$\text{to } 255_{10} \text{ (X'FF)} = 1\,1\,1\,1\,1\,1\,1\,1$$

Signed, the high-order bit is 0 for + (plus), 1 for - (minus), and the numbering range is as follows:

$$+127_{10} = 0\,1\,1\,1\,1\,1\,1\,1$$
$$\vdots$$
$$0_{10} = 0\,0\,0\,0\,0\,0\,0\,0$$
$$-\ 1_{10} = 1\,1\,1\,1\,1\,1\,1\,1$$
$$\vdots$$
$$-128_{10} = 1\,0\,0\,0\,0\,0\,0\,0$$

5.7.1.1 Arithmetic with Unsigned Data Bytes

Unsigned arithmetic uses the carry bit, but not the overflow bit. For example:

```
  X'2E  =  0 0 1 0 1 1 1 0
+ X'52  =  0 1 0 1 0 0 1 0
= X'80  =  1 0 0 0 0 0 0 0      CY/L = 0, no carry. Note that OV would be
                                set, but the program is not concerned.

  X'AE  =  1 0 1 0 1 1 1 0
+ X'52  =  0 1 0 1 0 0 1 0
= X'100 =  0 0 0 0 0 0 0 0      CY/L = 1. Note that OV would be cleared.
```

Consider the following subtraction using twos-complement arithmetic:

X'AE - X'52 = X'AE + ~X'52 + X'01 = X'AE + X'AD + X'01 = X'5C

```
  X'AE  =  1 0 1 0 1 1 1 0
+ X'AD  =  1 0 1 0 1 1 0 1
+ X'01  =  0 0 0 0 0 0 0 1
= X'5C  =  0 1 0 1 1 1 0 0      CY/L = 1 answer positive. Note that OV = 1
```

The code for implementing the above operation is as follows:

```
C4AE        LDI     0AE
03          SCL
FC52        CAI     052
C8F9        ST      ANS
```

X'52 - X'AE = X'52 + ~X'AE + X'01 = X'52 + X'51 + X'01 = X'A4 = -X'5C

```
          X'52  =  0 1 0 1 0 0 1 0
        + X'51  =  0 1 0 1 0 0 0 1
        + X'01  =  0 0 0 0 0 0 0 1
        = X'A4  =  1 0 1 0 0 1 0 0      CY/L = 0 answer negative so take twos
ones complement =  0 1 0 1 1 0 1 1      complement. Note OV = 1.
twos complement =  0 1 0 1 1 1 0 0
                =  -X'5C
```

The code for taking the twos complement is as follows:

```
03          SCL
01          XAE
C400        LDI     0
78          CAE
0000
```

Rules for addition and subtraction using unsigned data bytes are as follows:

1. Ignore the OV bit.
2. When adding, if CY/L is set, add 1 to the next high-order digit. CY/L is automatically an input to the add operation.
3. When subtracting, if CY/L is set, the answer is positive. If CY/L is cleared, the answer is negative and is present in its twos-complement form.

In multibyte arithmetic, the preceding three rules apply to the leftmost (terminal or high-order) byte. Between lower-order bytes, the CY/L bit is always treated as a carry into the low-order bit of the next byte:

```
  X'13E7 = 0 0 0 1 0 0 1 1   1 1 1 0 0 1 1 1
+ X'24C2 = 0 0 1 0 0 1 0 0   1 1 0 0 0 0 1 0
= X'38A9 = 0 0 1 1 1 0 0 0   1 0 1 0 1 0 0 1

                   CY/L = 1
                   Carry into next byte
```

5.7.1.2 Arithmetic with Signed Data Bytes

Since in signed arithmetic the high-order bit represents the sign, carries out of bit 7 different from carries into bit 7 represent overflow.

When performing signed arithmetic, the rules are essentially the same as for unsigned arithmetic with the following exceptions.

In a single-precision (one-byte) operation, or when operating upon the most significant byte in a multibyte operation, the overflow (OV) flag is set when an incorrect sign bit is generated as a result of the operation.

In performing multibyte arithmetic, the low-order byte should be processed first and any carries generated should be added into the next higher byte. This can be done automatically if the CY/L bit is not modified between the two operations.

5.7.2 Overflow and Carry/Link

The overflow bit is set whenever an add or complement-and-add operation causes a different carry bit into bit 7 from that out of bit 7; otherwise, it is not reset. For example:

```
  X'5A = 0 1 0 1 1 0 1 0 =  90
+ X'75 = 0 1 1 1 0 1 0 1 = 117
= X'CF = 1 1 0 0 1 1 1 1 = -49   signed twos complement

           Carry out of bit 6, but not out of 7 bit.
           OV set to 1.

  X'5A = 0 1 0 1 1 0 1 0 =  90
+ X'24 = 0 0 1 0 0 1 0 0 =  36
= X'7E = 0 1 1 1 1 1 1 0 = 126

           No carry out of bit 6, nor out of bit 7.
           OV set to 0.
```

The overflow bit is useful with signed arithmetic operations to indicate the generation of a result with an incorrect sign.

In addition to being used in rotate operations when specified, the carry/link bit is set whenever an add, decimal add, or complement-and-add operation causes a carry out of bit 7; otherwise, it is reset.

5.7.2.1 Add Operation with CY/L initially reset to 0

```
 X'B4 =    1 0 1 1 0 1 0 0  =  -76
+X'D6 =    1 1 0 1 0 1 1 0  =  -42
=X'8A =    1 0 0 0 1 0 1 0  = -118   signed twos complement
           |
          [1]
```

Carry out of bit 7; CY/L is set to 1.
(Note that in this case OV would be reset.
Ones are carried into and out of bit 7.)

```
 X'34 =    0 0 1 1 0 1 0 0  =   52
+X'56 =    0 1 0 1 0 1 1 0  =   86
= X'8A =   1 0 0 0 1 0 1 0  = -118   signed twos complement or 138 unsigned 8-bit binary
```

No carry out of bit 7; CY/L is set to 0.
(Note that in this case OV would be set.)

5.7.2.2 Decimal Add Operation with CY/L initially reset to 0

(Note positive integers assumed)

```
 X'52 =    0 1 0 1 0 0 1 0  =   52
+X'86 =    1 0 0 0 0 1 1 0  =   86
= X'38 =   0 0 1 1 1 0 0 0  =   38
```

Carry out of high-order digit; CY/L is set to 1.

5.7.2.3 Complement and Add Operation with CY/L initially set to 1

```
  X'34                =   0 0 1 1 0 1 0 0  =   52
+ ~ X'56  =  X'A9     =   1 0 1 0 1 0 0 1  =  -86
  = X'DE              =   1 1 0 1 1 1 1 0  =  -34
```

No carry out of bit 7; CY/L is set to 0.
(Note that in this case OV would be reset to 0.)

Chapter 6

MACROS

6.1 INTRODUCTION

Programming in simple assembly language enables a user to be as efficient with his microprocessor resources as his capabilities allow. With assembly language, the user can specify explicitly every detail of the program operation: indeed, he must specify every detail. Because of this, a program in assembly language often takes longer to write than the same program written in a high-level language, which fills in many details automatically according to its internal design. This design may or may not be compatible with either the language of the machine on which the high-level language operates or the user's problem. Ideally, the user would like a programming language that (1) most closely resembles the machine language, when it is appropriate, and (2) is in a form most suitable (natural) for expressing the user's problem. The language should fill in details whenever they are routine and should leave the user free to specify the details whenever they are crucial. This ideal can often be closely approximated by the use of a versatile programming tool known as macros.

Macros are a form of text replacement that provide an automatic code-generation capability completely under the user's control. With macros, a user can gradually build a library tailored to his application, and, with a library of macros oriented toward a particular application, a user who is not a software expert can produce efficient maching-language code; and an experienced user can significantly reduce his program development time.

6.2 BASIC MACRO CONCEPTS

The main use of macros is to insert assembly language statements into a source program, as shown in figure 6-1. In the example, the original source program contains a macro instruction, or macro call, named RR4. RR4 is a macro that rotates the Accumulator right four times. When the assembler processes RR4, it inserts the predefined sequence of assembly language from the macro definition named RR4 into the source program immediately after the point of call (RR4). The process of inserting the text of the macro definition into the source program is called macro expansion. The expanded macro then is processed as if it were part of the original source program. You will note that the macro call itself does not produce any maching language code. The directives used to define the limits of the macro definition (.MACRO and .ENDM) are explained in detail later in this chapter.

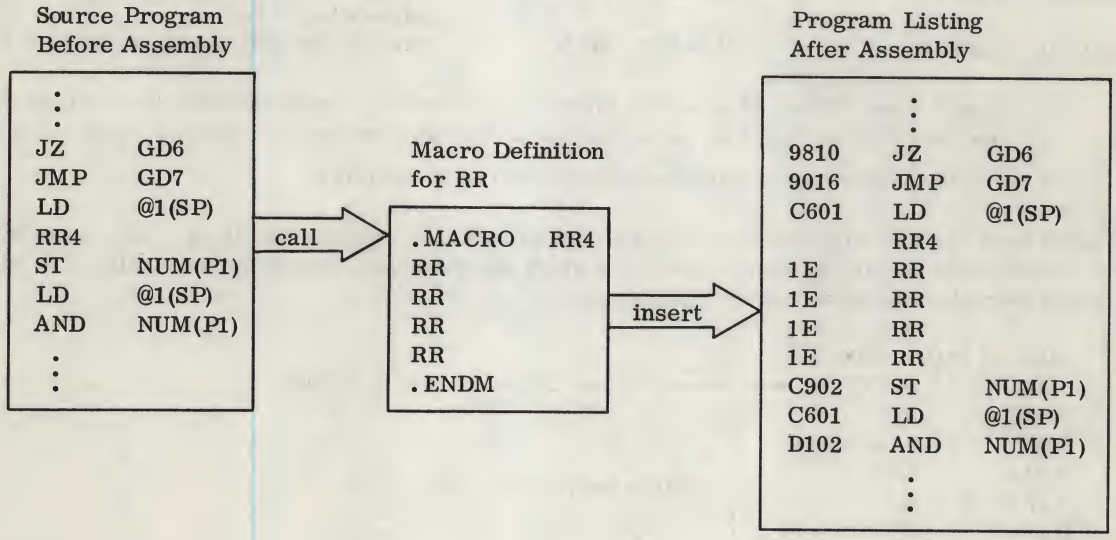


Figure 6-1. Statement Insertion

6.3 DEFINING A MACRO

Defining a macro means preparing the statements that constitute a macro definition. To define a macro, the following must be done:

- Give it a name.
- Declare any parameters to be used.
- Write the statements it contains.
- Establish its boundaries.

The following form is used to define a macro:

```
.MACRO   mname     [parameters]
           :
           :
         macro body
           :
           :
.ENDM
```

where:

(1) .MACRO is the directive that initiates the macro definition. Macros must be defined before their use. It is legal to define a macro with the same name as an already existing macro. The latest definition is always the operative one, but previous definitions are not discarded. They may be reactivated by using the .MDEL directive to delete the last macro definition (see 6.12).

(2) "mname" is the name of the macro (the name used to "call" the macro). The macro name must adhere to all rules for symbols (see 3.1.3.2).

(3) Parameters is the optional list of parameters used in the macro definition. The list of parameters must adhere to all the rules for expressions (see 3.1.4).

The following are examples of legal and illegal .MACRO directives:

Legal		Illegal		Reason Illegal
.MACRO	MAC,A,B	.MACRO	SUB,1	Special character is used in parameter.
.MACRO	$ADD,OP1,OP2	.MACRO	1MAC,C,D	First character in macro name is illegal.
.MACRO	LIST,$1	.MACRO	MACB,25	First character in parameter must be alphabetic or $.
.MACRO	MSG3	.MACRO	M$AC	Special character is used in macro name.

(4) Macro body consists of assembly language statements. The macro body may contain simple text (see below), text with parameters (see 6.5) and macro-time operators (see 6.5.3).

(5) .ENDM is the directive that terminates the macro definition.

The simplest form of a macro definition is one with no parameters or macro operators. The macro body is simply a sequence of assembly language statements which are substituted for each macro call. The following show several examples of simple macro definitions:

```
;SAVE STACK POINTER
.MACRO   SSTKP      <-- Begin the macro definition
XPAL     SP         \
ST       STKP (P1)   |
XPAL     SP          |  Macro body
XPAH     SP          |
ST       STKP + 1 (P1)|
XPAH     SP         /
.ENDM    <-- End the macro definition
```

```
;RESTORE STACK POINTER
.MACRO    RSTKP
LD        STKP (P1)
XPAL      SP
LD        STKP + 1 (P1)
XPAH      SP
.ENDM

;DELAY ONE MILLISECOND
.MACRO    DELAY
LDI       244
DLY       0
.ENDM
```

6.4 CALLING A MACRO

Once a macro has been defined, it then may be called. A macro is called by placing the macro name in the operation field of an assembly language statement, and the parameters in the operand field. The following form is used for a macro call:

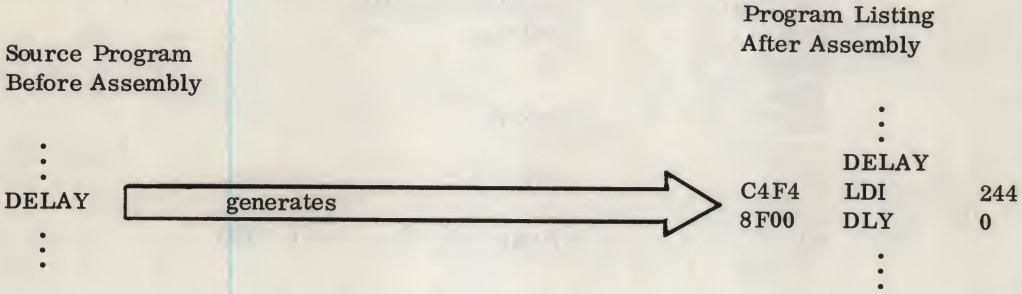
```
mname     [parameters]
```

where

(1) "mname" is the name previously assigned in the macro definition.

(2) "parameters" is the list of input parameters. When a macro is defined without parameters, the parameter list is omitted from the call.

A call to the delay macro, defined in 6.3, would be as follows:



6.5 USING PARAMETERS

The power of a macro can be increased tremendously through the use of the optional parameters. The parameters allow variable values to be declared when the macro is called.

6.5.1 Macro Definition

The delay macro previously illustrated could be made more powerful through the use of parameters. By making the delay constants depend on the call, the same macro can be used for a variety of delays. In this example, UCYCS is a formal parameter and can be replaced with any value at expansion time.

```
;DELAY UCYCS MICROCYCLES
.MACRO    DELAY2,UCYCS
LDI       UCYCS-13/2
DLY       0
.ENDM
```

Parameters need not be variables or numeric values, but can be any string. The following macro, for example, takes an ASCII string as input and generates a message string in memory suitable for input to the SC/MP firmware MESG routine.

```
                .MACRO      MSGSTR,LABEL,STRING
LABEL:          .ASCII      'STRING'
                .BYTE       0
                .ENDM
```

The following macro generates a call to the MESG routine with the name of the message as input.

```
                .MACRO      MESG,MSGNAM
                JS          P3,07EA7
                .DBYTE      MSGNAM
                .ENDM
```

Note the principle here: the hexadecimal firmware address is maintained centrally in the MESG macro, not scattered all over the code as the macro calls will be.

Finally, here are some macros of obvious utility in a SC/MP environment:

```
;LOAD POINTER IMMEDIATE
.MACRO          LPI,PTR,SYMB
LDI             H(SYMB)
XPAH            PTR
LDI             L(SYMB)
XPAL            PTR
.ENDM

;EXCHANGE POINTER WITH MEMORY
.MACRO          EXCHP,PTR,SYMB
LDI             L(SYMB)
XPAL            PTR
XAE
LDI             H(SYMB)
XPAH            PTR
ST              (PTR)           ;H(PTR)
LDE
ST              1(PTR)          ;L(PTR)
.ENDM

;PROGRAM HEADER MACRO
.MACRO PROGR,NAME,NUM,VER,DATE
.TITLE NAME,'NUM VER DATE'
P0=0
P1=1
P2=2
P3=3
PC=P0
STKPT=P2
.ENDM

.MACRO PUSH
ST              @-1(STKPT)
.ENDM

.MACRO PULL
LD              @1(STKPT)
.ENDM
```

6.5.2 Calling a Macro With Parameters

When parameters are included in a macro call, the following rules apply to the parameter list:

1. Commas or blanks delimit parameters.
2. Consecutive blanks are treated as a single delimiter.
3. A comma leading, following, or imbedded in a string of blanks is treated as a single delimiter.
4. A semicolon terminates the list and starts the comment field.
5. Quotes may be included as part of a parameter except as the first character of a parameter.
6. A parameter may be enclosed in quotes (') in which case the quotes are removed and the string is used as the parameter. This function is useful when blanks, commas, or semicolons are to be included in the parameter.
7. To include a quote in a quoted parameter, it must be entered as two consecutive quotes.
8. Missing or null parameters are treated as strings of length zero.

The following examples illustrate these rules:

NOTE

In these examples, for clarity, # indicates the parameter; strings are delimited by double primes (").

Macro Call	Parameter List		
MAC1 NAME NO ONE MAN	#1	"NAME"	
	#2	"NO"	
	#3	"ONE"	
	#4	"MAN"	
	#5,#6,. . .	" "	(NULL PARAMETER)
MAC2 MADAM 'I' 'M' ADAM	#1	"MADAM"	
	#2	"I'M"	
	#3	"ADAM"	
	#4,#5,. . .	" "	(NULL PARAMETER)
MAC3 'A MAN','A PLAN','A CANAL';PANAMA			
	#1	"A MAN"	
	#2	"A PLAN"	
	#3	"A CANAL"	
	#4,#5,. . .	" "	(NULL PARAMETER)
MAC4 POOR,,'IS IN A DROOP'	#1	"POOR"	
	#2	" "	(NULL PARAMETER)
	#3	"IS IN A DROOP"	
	#4,#5,. . .	" "	(NULL PARAMETER)

The following examples show some macro calls to macros defined in 6.3. These illustrate the parameter parsing and substitution.

Macro Call		Generated Code
DELAY2 100		LDI 100-13/2
		DLY 0
MSGSTR PROMPT 'ENTER VALUE'	PROMPT:	.ASCII 'ENTER VALUE'
		.BYTE 0

6.5.3 Parameters Referenced by Number

6.5.3.1 '#' — Number of Parameters

'#' is a macro operator that references the parameter list in the macro call. When used in an expression, it is replaced by the number of parameters in the macro call. The following .IF directive, for example, causes the conditional code to be expanded if there are more than 10 parameters in the macro call:

```
.IF     #  10
```

6.5.3.2 '#N' — Nth Parameter

When used in conjunction with a constant or variable, the '#' operator references individual parameters in the parameter list. The following example demonstrates how this function is used.

```
.MACRO   X
.BYTE    #1,#2,#2
.ENDM

X        3,5,2          generates          .BYTE    3,5,2
```

This relieves the need for naming each parameter in a long list and allows powerful macros to be defined using arbitrary numbers of parameters.

6.5.4 ' ^ ' — Concatenation

The ' ^ ' macro operator is used for concatenation. When found, the ' ^ ' is removed from the output string and the strings on each side of the operator are compressed together after parameter substitution. If a set variable is used with the ' ^ ' operator, it is converted to a hexadecimal number before being placed in to the output stream.

Example:

```
.MACRO    IMAGINARY,X
R^X:      .BYTE   0
I^X:      .BYTE   0
.ENDM
```

Another example of the use of this operator is shown in 6.9.3.

6.6 LOCAL SYMBOLS

```
.MLOC    symbol[,symbol...]                    [;comments]
```

When a label is defined within a macro, a duplicate definition results with the second and each subsequent call. The problem can be avoided by using the .MLOC directive to declare labels local to the macro definition.

Local symbols are replaced with unique names at expansion time with ZZxxxx, where xxxx is a 4-digit hexadecimal number. The user should avoid using his own labels of the above form as it may cause duplicate definition errors. The .MLOC directive may occur at any point in a macro definition, but it must precede the first occurrence of the symbols it declares local. If it does not, no error will be reported, but symbols used before the .MLOC will not be recognized as local.

6.7 CONDITIONAL EXPANSION

The versatility and the power of the macro assembler is enhanced by the conditional assembly directives. The conditional assembly directives (.IF, .ELSE and .ENDIF) from chapter 4 allow the user to generate different lines of code from the same macro simply by varying the parameter values used in the macro calls. Three relational operators are provided:

= (equal)
< (less than)
> (greater than)

6.7.1 .IF, .ELSE, .ENDIF Directives

When the macro assembler encounters a .IF directive within a macro expansion, it evaluates the relational operation that follows. If the expression is satisfied (evaluated greater than 0), the lines following the .IF are expanded until a .ELSE or a .ENDIF directive is encountered. If the expression is not satisfied (evaluated less than or equal to 0), only the lines from the .ELSE to the .ENDIF are expanded. See 4.5.11 for additional information on the conditional assembly directives.

Example:

```
;SHIFT THE CONTENTS OF RAM ADDRESS X
; RIGHT IF N > 0, LEFT OTHERWISE
.MACRO    SHIFT,X,N
LD        X
.IF       N > 0
SR
.ELSE
CCL
ADD       X
.ENDIF
ST        X
.ENDM
```

6.7.2 .IFC Directive

```
[label]          .IFC        string1 operator string2          [;comments]
```

The .IFC directive allows conditional assembly based on character strings rather than the value of an expression as in the .IF directive. String1 and String2 are the character strings to be compared. Operator is the relational operator between the strings. Two operators are allowed: EQ (equal) and NE (not equal). If the relational operator is satisfied, the lines following the .IFC are assembled until a .ELSE or a .ENDIF is encountered. The .ELSE and .ENDIF directives have the same effect with the .IFC directive as they do with the .IF directive.

The primary application of the .IFC is to compare a parameter value such as #3 against a specific string. For example:

```
.IFC #3 NE INTEGER
```

6.8 USEFUL DIRECTIVES

6.8.1 .SET Directive

```
[label]        .SET        symbol,expression                    [;comments]
```

The .SET directive is used to assign values to symbols (variables). A variable assigned a value with the .SET directive can be reassigned different values an arbitrary number of times. Set variables are useful during macro expansion to control macro-time looping and macro communication. To insure value correspondence between pass one and pass two of the assembler, all values in the expression must be defined before use in a .SET directive. If a value is not previously defined, an error is reported and a value of zero is returned.

The .SET directive provides what is needed to make a good DELAY macro which accepts an argument in milliseconds. We define:

```
MPMS=500                                ;MICROCYCLES PER MILLISECOND
;DELAY MS MILLISECONDS
;FORMULAS FROM ISP-8S/994Y  PAGE 4-24
.MACRO DELAY3,MS
.SET        MC,MS*MPMS-13
.SET        DLYDSP,MC/514               ;(M-13)/514
.SET        CAC,514*DLYDSP/2
.SET        CAC,MC/2-CAC                ;((M-13)-514(DISP))/2
LDI         CAC
DLY         DLYDSP
.ENDM
```

Of course, this macro would benefit from a little more attention to numerical analysis, but the idea is clear: forget the tables and let the assembler do the arithmetic; its not only easier, its more reliable.

6.8.2 .MDEL Directive

```
[label]        .MDEL       mname[,mname...]                     [;comments]
```

The .MDEL directive deletes macro definitions from the macro definition table and frees the buffer space used by the definitions.

Example:

```
.MDEL       DELAY,MAC1
```

6.8.3 .ERROR Directive

```
[label]        .ERROR      [string]                             [;comments]
```

The .ERROR directive generates an error message and an assembly error that is included in the error count at the end of the program. The directive is useful for parameter checking in macros. For example, the LPI macro defined in 6.5.1 will put out erroneous code if PTR=0, therefore, we should flag that condition.

```
;LOAD POINTER (OTHER THAN PC) IMMEDIATE
.MACRO    LPI, PTR, EXPR
.IF       PTR < 4
.IF       PTR > 0
.SET      SYMB, EXPR
LDI       H(SYMB)
XPAH      PTR
LDI       L(SYMB)
XPAL      PTR
.ELSE
  .IF     PTR=0
  .ERROR  'LDI WILL NOT WORK WITH THE PC'
  .ELSE
  .ERROR  'ILLEGAL POINTER VALUE'
  .ENDIF
.ENDIF
.ENDM
```

6.9 MACRO-TIME LOOPING

6.9.1 .DO and .ENDDO Directives

```
[label]     .DO       count     [;comments]
[label]     .ENDDO                [;comments]
```

Macro-time looping is facilitated through the .DO and .ENDDO directives. These directives are used to delimit a block of statements which are repeatedly assembled. The number of times the block will be assembled is specified on the .DO directive. Following is the format of a .DO-.ENDDO block:

```
.DO       count
            .
            .
            .
          source
            .
            .
            .
          .ENDDO
```

NOTE

.DO, .ENDDO, and .EXIT are defined only within a macro definition.

6.9.2 .EXIT Directive

```
[label]     .EXIT                 [;comments]
```

Early termination of looping in a .DO-.ENDDO block can be effected with the .EXIT directive.

This directive allows the current loop to finish and then terminates looping.

6.9.3 Examples of Macro-Time Loops

The following examples show the use of the .DO, .ENDDO, and .EXIT directives. The macro CTAB generates a constant table from 0 to MAX where MAX is a parameter of the macro call. Each word has a label DX: — where X is the value of the data word.

```
            .MACRO    CTAB,MAX
            .SET      X,0
            .DO       MAX+1
D^X:        .BYTE     X
            .SET      X,X+1
            .ENDDO
            .ENDM
```

Now a call of the form:

```
            CTAB      10
```

generates code equivalent to:

```
            .SET      X,0
D00:        .BYTE     X
            .SET      X,X+1
D01:        .BYTE     X
            .SET      X,X+1
D02:        .BYTE     X
              :
              :
            .SET      X,X+1
D09:        .BYTE     X
            .SET      X,X+1
D0A:        .BYTE     X
```

The macro MSGLST generates a call to the SC/MP firmware MESG routine with each of the parameters on the macro call. The parameter list may be any length.

```
            .MACRO    MSGLST
            .SET      X,0
            .DO       -1            ;SET FOR INFINITE LOOPING
            .SET      X,X+1
            .IF       X > #         ;CHECK IF NUMBER OF TIMES THRU LOOP (X)
                                    ;IS > NUMBER OF PARAMETERS CALLED BY MSGLST
            .EXIT                   ;YES
            .ELSE
            MESG      #X            ;NO, CALL MESG MACRO
            .ENDIF
            .ENDDO
            .ENDM
```

Now a call of the form:

```
            MSGLST    MSG1,MSG2
```

generates code equivalent to:

```
            JS        P3,07EA7
            .DBYTE    MSG1
            JS        P3,07EA7
            .DBYTE    MSG2
```

NOTE

Care must be taken when writing macros that generate a variable number of data words through the use of the .IF or the .DO directives. If the operands on these directives are forward references, their values change between pass 1 and pass 2 and the number of generated words may change. Should this be the case, all labels defined after the macro call that has changed values generate numerous assembly errors of the following form:

ERROR DUP. DEF.

6.10 NESTED MACRO CALLS

Nested macro calls are allowed, that is, a macro definition may contain a call to another macro. When a macro call is encountered during macro expansion, the state of the macro currently being expanded is saved and expansion begins on the nested macro. Upon completing expansion of the nested macro, expansion of the original macro continues. Depth of nesting allowed will depend on the parameter list sizes, but on the average about 10 levels of nesting will be allowed.

A logical extension of a nested macro call is a recursive macro call, that is a macro that calls itself. This is allowed, but care must be taken that an infinite loop is not generated.

The following macros illustrate the value of the nesting capability. The first macro is a utility macro used to load a pointer register.

```
;LOAD POINTER IMMEDIATE
.MACRO    LPI, PTR, SYMB
LDI       H(SYMB)
XPAH      PTR
LDI       L(SYMB)
XPAL      PTR
.ENDM
```

The next macro is named WRITE and it uses LPI to load the address of the write routine. MESG is the firmware address of print routine.

```
;WRITE A MESSAGE
.MACRO    WRITE, MESS
LPI       P3, MESG
XPPC      P3
.DBYTE    MESS
.ENDM
```

Now a call to the WRITE macro

```
WRITE     ENDMSG
```

generates the following code:

```
WRITE     ENDMSG
LDI       H(MESG)
XPAH      P3
LDI       L(MESG)
XPAL      P3
XPPC      P3
.DBYTE    ENDMSG
```

Of course, the JS pseudo instruction could have been used instead of a macro; the number of instructions generated is the same, but this simple case illustrates the concept.

6.11 NESTED MACRO DEFINITIONS

A macro definition can be nested within another macro. Such a macro is not defined until the outer macro is expanded and the nested .MACRO statement is executed. This allows the creation of special-purpose macros based on the outer macros parameters and, when used with the .MDEL directive, allows a macro to be defined only within the range of the macro that uses it.

Chapter 7

ASSEMBLER INPUT/OUTPUT FORMATS

The information in this chapter is common to all the SC/MP cross assemblers.

The SC/MP cross assembler programs assemble a source program on a host computer for subsequent execution by a SC/MP Microprocessor. Detailed operation instructions on the SC/MP cross assembler programs are contained in the installation and operating instructions for the assemblers.

Each SC/MP cross assembler requires the following minimum complement of peripherals: a source input unit, a list output unit, and a binary output unit. In addition, some cross assemblers require a scratch unit that can be rewound for multipass processing.

The input and output files required by the assemblers are listed below and explained in the following paragraphs.

Function	File Format	Logical Record Length
Source File (Input)	Sequential	80 bytes
Program Listing File (Output)	Sequential	121 bytes
Load Module File (Output)	Binary	36 bytes

7.1 SOURCE FILE (INPUT)

The Source File may be input from paper tape, cards, or diskette.

7.2 PROGRAM LISTING FILE (OUTPUT)

The Program Listing file contains ANS FORTRAN carriage-control characters.

At the end of the program listing, a symbol table is produced, a message is printed noting the number of errors discovered by the assembler program, and the source and the object checksums are printed.

7.3 LOAD MODULE (OUTPUT)

The Load Module (LM) file contains loading information and object code produced from the source statements. The LM file is an unformatted file composed of a sequence of records, each containing up to 36 bytes. The representation of the records depends on the storage medium. There are three types of LM records:

Title Record (one per LM file)
Data Record (variable number per LM file)
End Record (one per LM file)

The records are produced in the sequence shown in figure 7-1. Independent of the record type, the first two bytes in each record always have the same interpretation. The first byte specifies the record type (bits 7 and 6) and the length of the record body (bits 5 through 0). The second byte contains a checksum for error detection. The checksum is formed by taking the arithmetic sum (modulo 2^8) of all the bytes in the record body.

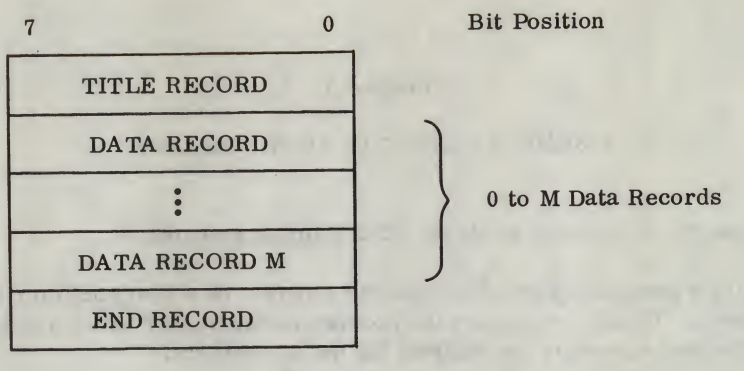
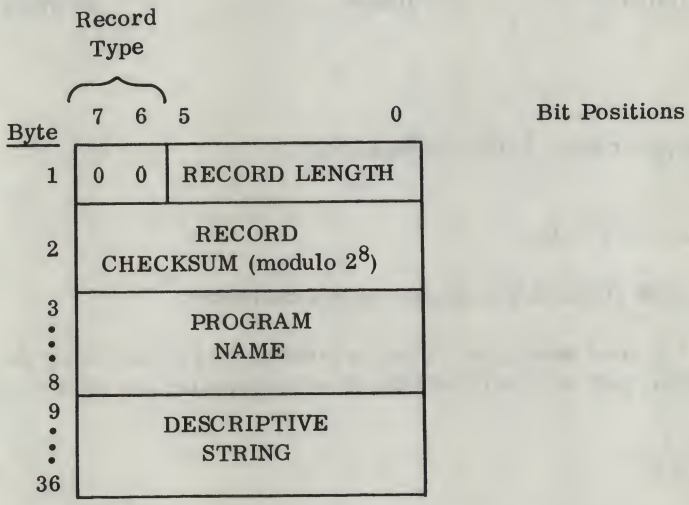


Figure 7-1. LM File Format

7.3.1 Title Record

The Title Record identifies the load module by name and, optionally, by a descriptive character string. These two items are supplied by the last . TITLE Directive statement in the source program. If the . TITLE Directive is not included, a default name (MAINPR) is used. If the default program name is assigned, the descriptive string is empty. Figure 7-2 illustrates the format of the Title Record.



NOTES

1. The program name and descriptive string are composed of 7-bit ASCII characters. The strings are right-justified with a zero-fill at the end.
2. Only the first 28 characters in the descriptive string (of the source statement) are used in the title record.

Figure 7-2. Title Record Format

7.3.2 Data Record

The Data Records contain the actual data and instruction bytes to be loaded into memory. Each data record contains the load address of the initial data byte of the record. Each time a discontinuity (empty area or change-of-page) occurs in a program, the current record is terminated and outputted, and a new record is initiated. Figure 7-3 illustrates the format of the Data Record.

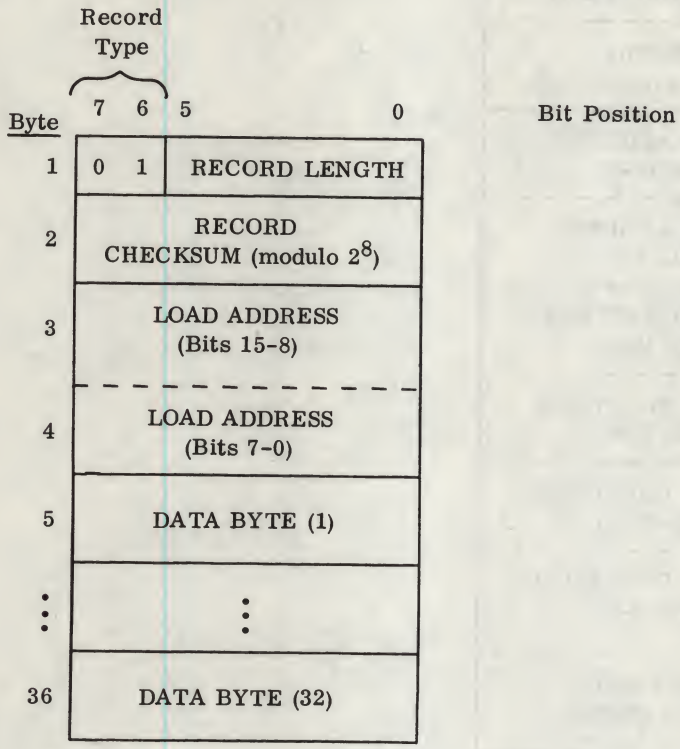


Figure 7-3. Data Record Format

7.3.3 End Record

The End Record marks the end of the LM file and specifies an entry address for the load module. The format of the End Record is illustrated in figure 7-4.

The source checksum represents the sum (modulo 2^{16}) of all the characters, taken one at a time, in a program source file. The sum is printed on the program listing following the symbol table printout.

The object sum represents the sum (modulo 2^{16}) of all the individual record checksums of the LM. This sum is also printed on the program listing following the symbol table.

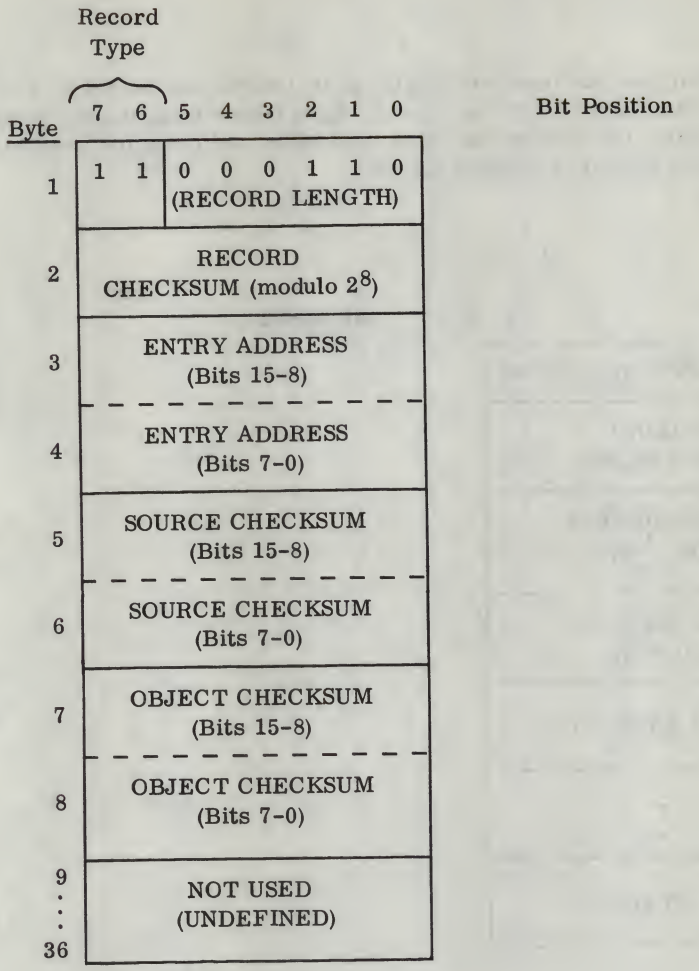


Figure 7-4. End Record Format

Appendix A

ASCII CHARACTER SET

Table A-1 contains the 7-bit hexadecimal code for each character in the ASCII character set. The printable characters in this set may be set up as program data by use of the .ASCII directive. The remaining characters may be set up in hexadecimal constants with a .BYTE or .DBYTE directive. Table A-2 contains the legend for nonprintable characters.

Table A-1. ASCII Character Set in Hexadecimal Representation

Character	7-Bit Hexadecimal Number	Character	7-Bit Hexadecimal Number	Character	7-Bit Hexadecimal Number	Character	7-Bit Hexadecimal Number
NUL	00	SP	20	@	40	`	60
SOH	01	!	21	A	41	a	61
STX	02	"	22	B	42	b	62
ETX	03	#	23	C	43	c	63
EOT	04	$	24	D	44	d	64
ENQ	05	%	25	E	45	e	65
ACK	06	&	26	F	46	f	66
BEL	07	'	27	G	47	g	67
BS	08	(	28	H	48	h	68
HT	09	)	29	I	49	i	69
LF	0A	*	2A	J	4A	j	6A
VT	0B	+	2B	K	4B	k	6B
FF	0C	,	2C	L	4C	l	6C
CR	0D	-	2D	M	4D	m	6D
SO	0E	.	2E	N	4E	n	6E
SI	0F	/	2F	O	4F	o	6F
DLE	10	0	30	P	50	p	70
DC1	11	1	31	Q	51	q	71
DC2	12	2	32	R	52	r	72
DC3	13	3	33	S	53	s	73
DC4	14	4	34	T	54	t	74
NAK	15	5	35	U	55	u	75
SYN	16	6	36	V	56	v	76
ETB	17	7	37	W	57	w	77
CAN	18	8	38	X	58	x	78
EM	19	9	39	Y	59	y	79
SUB	1A	:	3A	Z	5A	z	7A
ESC	1B	;	3B	[	5B		7B
FS	1C	<	3C	\	5C		7C
GS	1D	=	3D	]	5D	ALT	7D
RS	1E	>	3E	↑	5E	ESC	7E
US	1F	?	3F	←	5F	DEL, RUBOUT	7F

Table A-2. Legend for Nonprintable Characters

Character	Definition
NUL	NULL
SOH	START OF READING; ALSO START OF MESSAGE
STX	START OF TEXT; ALSO EOA, END OF ADDRESS
ETX	END OF TEXT; ALSO EOM, END OF MESSAGE
EOT	END OF TRANSMISSION (END)
ENQ	ENQUIRY (ENQRY); ALSO WRU
ACK	ACKNOWLEDGE. ALSO RU
BEL	RINGS THE BELL
BS	BACKSPACE
HT	HORIZONTAL TAB
LF	LINE FEED OR LINE SPACE (NEW LINE): ADVANCES PAPER TO NEXT LINE BEGINNING OF LINE
VT	VERTICAL TAB (VTAB)
FF	FORM FEED TO TOP OF NEXT PAGE (PAGE)
CR	CARRIAGE RETURN

Character	Definition
SO	SHIFT OUT
SI	SHIFT IN
DLE	DATA LINK ESCAPE
DC1	DEVICE CONTROL 1
DC2	DEVICE CONTROL 2
DC3	DEVICE CONTROL 3
DC4	DEVICE CONTROL 4
NAK	NEGATIVE ACKNOWLEDGE
SYN	SYNCHRONOUS IDLE (SYNC)
ETB	END OF TRANSMISSION BLOCK
CAN	CANCEL (CANCL)
EM	END OF MEDIUM
SUB	SUBSTITUTE
ESC	ESCAPE. PREFIX
FS	FILE SEPARATOR
GS	GROUP SEPARATOR
RS	RECORD SEPARATOR
US	UNIT SEPARATOR
SP	SPACE

Appendix B

INDEX OF INSTRUCTIONS

Table B-1. Opcode Index of Instructions

Opcode	Mnemonic	Operation	μcycles	Bytes/ Instruction	Page
00	HALT	Pulse H-flag	8	1	4-19
01	XAE	Exchange AC and Extension	7	1	4-14
02	CCL	Clear Carry/Link	5	1	4-20
03	SCL	Set Carry/Link	6	1	4-20
04	DINT	Disable Interrupts	6	1	4-21
05	IEN	Enable Interrupts	6	1	4-20
06	CSA	Copy Status to AC	5	1	4-21
07	CAS	Copy AC to Status	6	1	4-21
08	NOP	No Operation	5	1	4-22
19	SIO	Serial Input/Output	5	1	4-17
1C	SR, SHR	Shift Right	5	1	4-18
1D	SRL	Shift Right with CY/L	5	1	4-18
1E	RR, ROR	Rotate Right	5	1	4-18
1F	RRL	Rotate Right with CY/L	5	1	4-19
30	XPAL	Exchange Pointer Low	8	1	4-16
34	XPAH	Exchange Pointer High	8	1	4-17
3C	XPPC	Exchange Pointer with PC	7	1	4-17
40	LDE	Load AC from Extension	6	1	4-14
50	ANE	AND AC with Extension	6	1	4-14
58	ORE	OR AC with Extension	6	1	4-14
60	XRE	Exclusive-OR AC with Extension	6	1	4-15
68	DAE	Decimal Add AC and Extension	11	1	4-15
70	ADE	Add AC and Extension	7	1	4-15
78	CAE	Complement and Add Extension to AC	8	1	4-16
8FXX	DLY	Delay	13-131593	2	4-22
90XX	JMP	Jump	11	2	4-12
94XX	JP	Jump If Positive	9,11	2	4-13
98XX	JZ	Jump If Zero	9,11	2	4-13
9CXX	JNZ	Jump If Not Zero	9,11	2	4-13
A8XX	ILD	Increment and Load	22	2	4-8
B8XX	DLD	Decrement and Load	22	2	4-9
C0XX	LD	Load	18	2	4-5
C4XX	LDI, LI	Load Immediate	10	2	4-9
C8XX	ST	Store	18	2	4-6
D0XX	AND	AND	18	2	4-6
D4XX	ANI	AND Immediate	10	2	4-10
D8XX	OR	OR	18	2	4-6
DCXX	ORI	OR Immediate	10	2	4-10
E0XX	XOR	Exclusive-OR	18	2	4-6
E4XX	XRI	Exclusive-OR Immediate	10	2	4-10
E8XX	DAD, DECA	Decimal Add	23	2	4-7
ECXX	DAI	Decimal Add Immediate	15	2	4-11
F0XX	ADD	Add	19	2	4-7
F4XX	ADI	Add Immediate	11	2	4-11
F8XX	CAD	Complement and Add	20	2	4-8
FCXX	CAI	Complement and Add Immediate	12	2	4-11

Note: XX = disp or data

Table B-2. Mnemonic Index of Instructions

Mnemonic	Opcode	Description	μcycles	Byte/ Instruction	Page
ADD	F0XX	Add	19	2	4-7
ADE	70	Add AC and Extension	7	1	4-15
ADI	F4XX	Add Immediate	11	2	4-11
AND	D0XX	AND	18	2	4-6
ANE	50	AND AC with Extension	6	1	4-14
ANI	D4XX	AND Immediate	10	2	4-10
CAD	F8XX	Complement and Add	20	2	4-8
CAE	78	Complement and Add Extension to AC	8	1	4-16
CAI	FCXX	Complement and Add Immediate	12	2	4-11
CAS	07	Copy AC to Status	6	1	4-21
CCL	02	Clear Carry/Link	5	1	4-20
CSA	06	Copy Status to AC	5	1	4-21
DAD	E8XX	Decimal Add	23	2	4-7
DAE	68	Decimal Add AC and Extension	11	1	4-15
DAI	ECXX	Decimal Add Immediate	15	2	4-11
DINT	04	Disable Interrupts	6	1	4-21
DLD	B8XX	Decrement and Load	22	2	4-9
DLY	8FXX	Delay	13-131593	2	4-22
HALT	00	Pulse H-flag	8	1	4-19
IEN	05	Enable Interrupts	6	1	4-20
ILD	A8XX	Increment and Load	22	2	4-8
JMP	90XX	Jump	11	2	4-12
JNZ	9CXX	Jump If Not Zero	9, 11	2	4-13
JP	94XX	Jump If Positive	9, 11	2	4-13
JZ	98XX	Jump If Zero	9, 11	2	4-13
LD	C0XX	Load	18	2	4-5
LDE	40	Load AC from Extension	6	1	4-14
LDI	C4XX	Load Immediate	10	2	4-9
LI	C4XX	Load Immediate	5	2	4-9
NOP	08	No Operation	5	1	4-22
OR	D8XX	OR	18	2	4-6
ORE	58	OR AC with Extension	6	1	4-14
ORI	DCXX	OR Immediate	10	2	4-10
ROR	1E	Rotate Right	10	1	4-18
RR	1E	Rotate Right	5	1	4-18
RRL	1F	Rotate Right with Link	5	1	4-19
SCL	03	Set Carry/Link	5	1	4-20
SHR	1C	Shift Right	5	1	4-18
SIO	19	Serial Input/Output	5	1	4-17
SR	1C	Shift Right	5	1	4-18
SRL	1D	Shift Right with Link	5	2	4-18
ST	C8XX	Store	18	1	4-6
XAE	01	Exchange AC and Extension	7	2	4-14
XOR	E0XX	Exclusive-OR	18	1	4-6
XPAH	34	Exchange Pointer High	8	1	4-17
XPAL	30	Exchange Pointer Low	8	1	4-16
XPPC	3C	Exchange Pointer with PC	7	1	4-17
XRE	60	Exclusive-OR AC with Extension	6	1	4-15
XRI	E4XX	Exclusive-OR Immediate	10	2	4-10

Note: XX = disp or data

Table B-3. Numeric Index of Single-Byte Instruction Codes

Read down then right

Description	Mnemonic and Assembler Format		PC	P1	P2	P3
Halt	HALT	00				
Exchange AC and Extension	XAE	01				
Clear Carry/Link	CCL	02				
Set Carry/Link	SCL	03				
Disable Interrupt	DINT	04				
Enable Interrupt	IEN	05				
Copy Status to AC	CSA	06				
Copy AC to Status	CAS	07				
No Operation	NOP	08				
Serial Input/Output	SIO	19				
Shift Right	SR, SHR	1C				
Shift Right with Link	SRL	1D				
Rotate Right	RR, ROR	1E				
Rotate Right with Link	RRL	1F				
Exchange Pointer Low	XPAL ptr		30	31	32	33
Exchange Pointer High	XPAH ptr		34	35	36	37
Exchange Pointer with PC	XPPC ptr		3C	3D	3E	3F
Load AC from Extension	LDE	40				
AND AC with Extension	ANE	50				
OR AC with Extension	ORE	58				
Exclusive-OR AC with Extension	XRE	60				
Decimal Add AC and Extension	DAE	68				
Add AC and Extension	ADE	70				
Complement and Add Extension to AC	CAE	78				

Table B-4. Numeric Index of Double-Byte Instruction Codes

Read down and then right.

Description	Mnemonic and Assembler Format		PC REL	Indexed			Immediate	Auto-Indexed			
				P1 REL	P2 REL	P3 REL		P1 REL	P2 REL	P3 REL	
Delay	DLY data	8FXX									XX = 0 to 255 (unsigned)
Jump	JMP address		90XX	91XX	92XX	93XX					XX = -128 to +127
Jump	JMP disp(ptr)										
Jump If Positive	JP address		94XX	95XX	96XX	97XX					
Jump If Positive	JP disp(ptr)										
Jump If Zero	JZ address		98XX	99XX	9AXX	9BXX					
Jump If Zero	JZ disp(ptr)										
Jump If Not Zero	JNZ address		9CXX	9DXX	9EXX	9FXX					
Jump If Not Zero	JNZ disp(ptr)										
Increment and Load	ILD address		A8XX	A9XX	AAXX	ABXX					
Increment and Load	ILD disp(ptr)										
Decrement and Load	DLD address		B8XX	B9XX	BAXX	BBXX					
Decrement and Load	DLD disp(ptr)										
Load	LD address		C0XX	C1XX	C2XX	C3XX	C4XX	C5XX	C6XX	C7XX	
Load	LD disp(ptr)										
Load Immediate	LDI, LI data										
Load (auto-indexed)	LD @disp(ptr)										
Store	ST address		C8XX	C9XX	CAXX	CBXX		CDXX	CEXX	CFXX	
Store	ST disp(ptr)										
Store (auto-indexed)	ST @disp(ptr)										
AND	AND address		D0XX	D1XX	D2XX	D3XX	D4XX	D5XX	D6XX	D7XX	
AND	AND disp(ptr)										
AND Immediate	ANI data										
AND (auto-indexed)	AND @disp(ptr)										
OR	OR address		D8XX	D9XX	DAXX	DBXX	DCXX	DDXX	DEXX	DFXX	
OR	OR disp(ptr)										
OR Immediate	ORI data										
OR (auto-indexed)	OR @disp(ptr)										
Exclusive-OR	XOR address		E0XX	E1XX	E2XX	E3XX	E4XX	E5XX	E6XX	E7XX	
Exclusive-OR	XOR disp(ptr)										
Exclusive-OR Immediate	XRI data										
Exclusive-OR (auto-indexed)	XOR @disp(ptr)										

Table B-4. Numeric Index of Double-Byte Instruction Codes (Continued)

Read down and then right.

Description	Mnemonic and Assembler Format		PC REL	Indexed			Immediate	Auto-Indexed			
				P1 REL	P2 REL	P3 REL		P1 REL	P2 REL	P3 REL	
Decimal Add	DAD, DECA	address	E8XX	E9XX	EAXX	EBXX	ECXX	EDXX	EEXX	EFXX	XX = 0 to 99 (unsigned BCD)
Decimal Add	DAD, DECA	disp(ptr)									
Decimal Add Immediate	DAI	data									
Decimal Add (auto-indexed)	DAD, DECA	@disp(ptr)									
Add	ADD	address	F0XX	F1XX	F2XX	F3XX	F4XX	F5XX	F6XX	F7XX	XX = -128 to +127
Add	ADD	disp(ptr)									
Add Immediate	ADI	data									
Add (auto-indexed)	ADD	@disp(ptr)									
Complement and Add	CAD	address	F8XX	F9XX	FAXX	FBXX	FCXX	FDXX	FEXX	FFXX	
Complement and Add	CAD	disp(ptr)									
Complement and Add Immediate	CAI	data									
Complement and Add (auto-indexed)	CAD	@disp(ptr)									

Appendix C

INSTRUCTION EXECUTION TIMES

Instruction	Read Cycles	Write Cycles	Total Microcycles
ADD	3	0	19
ADE	1	0	7
ADI	2	0	11
AND	3	0	18
ANE	1	0	6
ANI	2	0	10
CAD	3	0	20
CAE	1	0	8
CAI	2	0	12
CAS	1	0	6
CCL	1	0	5
CSA	1	0	5
DAD, DECA	3	0	23
DAE	1	0	11
DAI	2	0	15
DINT	1	0	6
DLD	3	1	22
DLY	2	0	13-131593
HALT	2	0	8
IEN	1	0	6
ILD	3	1	22
JMP	2	0	11
JNZ	2	0	9, 11 for Jump

Instruction	Read Cycles	Write Cycles	Total Microcycles
JP	2	0	9, 11 for Jump
JZ	2	0	9, 11 for Jump
LD	3	0	18
LDE	1	0	6
LDI, LI	2	0	10
NOP	1	0	5
OR	3	0	18
ORE	1	0	6
ORI	2	0	10
RR, ROR	1	0	5
RRL	1	0	5
SCL	1	0	5
SIO	1	0	5
SR, SHR	1	0	5
SRL	1	0	5
ST	2	1	18
XAE	1	0	7
XOR	3	0	18
XPAH	1	0	8
XPAL	1	0	8
XPPC	1	0	7
XRE	1	0	6
XRI	2	0	10

If slow memory is being used, the appropriate delay should be added for each read or write cycle.

Appendix D

DIRECTIVES

Directive	Mnemonic	Operands
Title	.TITLE	symbol[,string]
End	.END	[address]
Local	.LOCAL	
List	.LIST	immediate
Space	.SPACE	immediate
Page	.PAGE	[string]
ASCII	.ASCII	string[,string...]
Byte	.BYTE	expression[,expression...]
Double Byte	.DBYTE	expression[,expression...]
Address	.ADDR	expression[,expression...]
Form	.FORM	symbol,exp[(esp)][,exp[(exp)]]...
Set	.SET	symbol,expression
Conditional Assembly	.IF	expression
	.ELSE	
	.ENDIF	
Macro *	.MACRO	mname[parameters]
End Macro *	.ENDM	
Macro Local *	.MLOC	symbol[,symbol...]
Do *	.DO	count
End Do *	.ENDDO	
Exit Do *	.EXIT	
Conditional Assembly	.IFC	string1 operator string2 (2)
Error	.ERROR	string
Delete Macro	.MDEL	mname[,mname...]

* Cannot be used outside a macro definition.

NOTES: 1. The .FORM directive is not available in the SC/MP (PACE) Cross Assembler.

2. In the .IFC directive, operator means a relational operator and may be either "EQ" for equal or "NE" for not equal.

Appendix E

PROGRAMMERS CHECKLIST

The following list of items is suggested for desk-checking a program before assembly.

1. Is the source program terminated by an . END Directive?
2. Is each label in the program terminated by a colon (:)?
3. Is each comment in the program preceded by a semi-colon (;)?
4. Is each string constant in the program set off on both ends by a prime (')?
5. Are all hexadecimal constants preceded by either X' or 0 (zero)?
6. For each .IF Directive in the program, is there a corresponding .ENDIF?
7. Are any symbols defined by two-level forward references? This is illegal.
8. Do transfer address operands consider memory page structure and PC pre-incrementation (before instruction fetch)?
9. Are the jumps relative to the current location specified in bytes?
10. For each .MACRO directive in the program, is there a corresponding .ENDM?
11. For each .DO directive in the program, is there a corresponding .ENDDO?

Appendix F

CONVERSION TABLES

Table F-1. Positive Powers of Two

n	2^n	n	2^n
1	2	51	22517 99813 68524 8
2	4	52	45035 99627 37049 6
3	8	53	90071 99254 74099 2
4	16	54	18014 39850 94819 84
5	32	55	36028 79701 89639 68
6	64	56	72057 59403 79279 36
7	128	57	14411 51880 75855 872
8	256	58	28823 03761 51711 744
9	512	59	57646 07523 03423 488
10	1024	60	11529 21504 60684 6976
11	2048	61	23058 43009 21369 3952
12	4096	62	46116 86018 42738 7904
13	8192	63	92233 72036 85477 5808
14	16384	64	18446 74407 37095 51616
15	32768	65	36893 48814 74191 03232
16	65536	66	73786 97629 48382 06464
17	13107 2	67	14757 39525 89676 41292 8
18	26214 4	68	29514 79051 79352 82585 6
19	52428 8	69	59029 58103 58705 65171 2
20	10485 76	70	11805 91620 71741 13034 24
21	20971 52	71	23611 83241 43482 26068 48
22	41943 04	72	47223 66482 86964 52136 96
23	83886 08	73	94447 32965 73929 04273 92
24	16777 216	74	18889 46593 14785 80854 784
25	33554 432	75	37778 93186 29571 61709 568
26	67108 864	76	75557 86372 59143 23419 136
27	13421 7728	77	15111 57274 51828 64683 8272
28	26843 5456	78	30223 14549 03657 29367 6544
29	53687 0912	79	60446 29098 07314 58735 3088
30	10737 41824	80	12089 25819 61462 91747 06176
31	21474 83648	81	24178 51639 22925 83494 12352
32	42949 67296	82	48357 03278 45851 66988 24704
33	85899 34592	83	96714 06556 91703 33976 49408
34	17179 86918 4	84	19342 81311 38340 66795 29881 6
35	34359 73836 8	85	38685 62622 76681 33590 59763 2
36	68719 47673 6	86	77371 25245 53362 67181 19526 4
37	13743 89534 72	87	15474 25049 10672 53436 23905 28
38	27487 79069 44	88	30948 50098 21345 06872 47810 56
39	54975 58138 88	89	61897 00196 42690 13744 95621 12
40	10995 11627 776	90	12379 40039 28538 02748 99124 224
41	21990 23255 552	91	24758 80078 57076 05497 98248 448
42	43980 46511 104	92	49517 60157 14152 10995 96496 896
43	87960 93022 208	93	99035 20314 28304 21991 92993 792
44	17592 18604 4416	94	19807 04062 85660 84398 38598 7584
45	35184 37208 8832	95	39614 08125 71321 68796 77197 5168
46	70368 74417 7664	96	79228 16251 42643 37593 54395 0336
47	14073 74883 55328	97	15845 63250 28528 67518 70879 00672
48	28147 49767 10656	98	31691 26500 57057 35037 41758 01344
49	56294 99534 21312	99	63382 53001 14114 70074 83516 02688
50	11258 99906 84262 4	100	12676 50600 22822 94014 96703 20537 6
		101	25353 01200 45645 88029 93406 41075 2

Table F-2. Negative Powers of Two

n	2^{-n}									
0	1.0									
1	0.5									
2	0.25									
3	0.125									
4	0.0625									
5	0.03125									
6	0.01562	5								
7	0.00781	25								
8	0.00390	625								
9	0.00195	3125								
10	0.00097	65625								
11	0.00048	82812	5							
12	0.00024	41406	25							
13	0.00012	20703	125							
14	0.00006	10351	5625							
15	0.00003	05175	78125							
16	0.00001	52587	89062	5						
17	0.00000	76293	94531	25						
18	0.00000	38146	97265	625						
19	0.00000	19073	48632	8125						
20	0.00000	09536	74316	40625						
21	0.00000	04768	37158	20312	5					
22	0.00000	02384	18579	10156	25					
23	0.00000	01192	09289	55078	125					
24	0.00000	00596	04644	77539	0625					
25	0.00000	00298	02322	38769	53125					
26	0.00000	00149	01161	19384	76562	5				
27	0.00000	00074	50580	59692	38281	25				
28	0.00000	00037	25290	29846	19140	625				
29	0.00000	00018	62645	14923	09570	3125				
30	0.00000	00009	31322	57461	54785	15625				
31	0.00000	00004	65661	28730	77392	57812	5			
32	0.00000	00002	32830	64365	38696	28906	25			
33	0.00000	00001	16415	32182	69348	14453	125			
34	0.00000	00000	58207	66091	34674	07226	5625			
35	0.00000	00000	29103	83045	67337	03613	28125			
36	0.00000	00000	14551	91522	83668	51806	64062	5		
37	0.00000	00000	07275	95761	41834	25903	32031	25		
38	0.00000	00000	03637	97880	70917	12951	66015	625		
39	0.00000	00000	01818	98940	35458	56475	83007	8125		
40	0.00000	00000	00909	49470	17729	28237	91503	90625		
41	0.00000	00000	00454	74735	08864	64118	95751	95312	5	
42	0.00000	00000	00227	37367	54432	32059	47875	97656	25	
43	0.00000	00000	00113	68683	77216	16029	73937	98828	125	
44	0.00000	00000	00056	84341	88608	08014	86968	99414	0625	
45	0.00000	00000	00028	43170	94304	04007	43484	49707	03125	
46	0.00000	00000	00014	21085	47152	02003	71742	24853	51562	5
47	0.00000	00000	00007	10542	73576	01001	85871	12426	75781	25
48	0.00000	00000	00003	55271	36788	00500	92935	56213	37890	625
49	0.00000	00000	00001	77635	68394	00250	46467	78106	68945	3125
50	0.00000	00000	00000	88817	84197	00125	23233	89053	34472	65625

Table F-3. Integer Conversion Table

POWERS OF 16

Example: $268{,}435{,}456_{10} = (2.68435456 \times 10^8)_{10} = 1000\ 0000_{16} = (10^7)_{16}$

16^n							n
						1	0
						16	1
						256	2
					4	096	3
					65	536	4
				1	048	576	5
				16	777	216	6
				268	435	456	7
			4	294	967	296	8
			68	719	476	736	9
		1	099	511	627	776	10 = A
		17	592	186	044	416	11 = B
		281	474	976	710	656	12 = C
	4	503	599	627	370	496	13 = D
	72	057	594	037	927	936	14 = E
1	152	921	504	606	846	976	15 = F

Decimal Values

The SC/MP microprocessor maintains negative numbers in twos-complement form. To convert a number in hexadecimal notation to its twos-complement equivalent, subtract the number from hexadecimal 2^n, where "n" is the number of binary bits in the computer word. For an 8-bit byte, "n" is 8, and 2^n is 1 0000 0000 (binary) or 100 (hex).

Thus, the negative of 1C is:

```
 100
  1C
----
  E4
```

A hexadecimal number will be negative in the SC/MP microprocessor if the left-most digit is 8, 9, A, B, C, D, E, or F (because all of these groupings start with a one). Thus, the twos-complement of C7 is

```
 100
  C7
----
  39
```

Perhaps an easier way to find the twos-complement of a hexadecimal number is first to take the ones-complement of the number; the ones-complement plus one is the twos-complement. The ones-complement of a number is its inverted form; simply exchange its ones for zeros, and its zeros for ones. Thus,

hexadecimal	binary equivalent	ones-complement
C7	1100 0111	0011 1000

ones-complement + 1

```
 0011 1000
      +1
-----------
 0011 1001
```

Hex twos-complement of C7 → 3 9

Table F-4. Hexadecimal and Decimal Fraction Conversion

1		2		3		4	
HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL EQUIVALENT
.0	.0000	.00	.0000 0000	.000	.0000 0000 0000	.0000	.0000 0000 0000 0000
.1	.0625	.01	.0039 0625	.001	.0002 4414 0625	.0001	.0000 1525 8789 0625
.2	.1250	.02	.0078 1250	.002	.0004 8828 1250	.0002	.0000 3051 7578 1250
.3	.1875	.03	.0117 1875	.003	.0007 3242 1875	.0003	.0000 4577 6367 1875
.4	.2500	.04	.0156 2500	.004	.0009 7656 2500	.0004	.0000 6103 5156 2500
.5	.3125	.05	.0195 3125	.005	.0012 2070 3125	.0005	.0000 7629 3945 3125
.6	.3750	.06	.0234 3750	.006	.0014 6484 3750	.0006	.0000 9155 2734 3750
.7	.4375	.07	.0273 4375	.007	.0017 0898 4375	.0007	.0001 0681 1523 4375
.8	.5000	.08	.0312 5000	.008	.0019 5312 5000	.0008	.0001 2207 0312 5000
.9	.5625	.09	.0351 5625	.009	.0021 9726 5625	.0009	.0001 3732 9101 5625
.A	.6250	.0A	.0390 6250	.00A	.0024 4140 6250	.000A	.0001 5258 7890 6250
.B	.6875	.0B	.0429 6875	.00B	.0026 8554 6875	.000B	.0001 6784 6679 6875
.C	.7500	.0C	.0468 7500	.00C	.0029 2968 7500	.000C	.0001 8310 5468 7500
.D	.8125	.0D	.0507 8125	.00D	.0031 7382 8125	.000D	.0001 9836 4257 8125
.E	.8750	.0E	.0546 8750	.00E	.0034 1796 8750	.000E	.0002 1362 3046 8750
.F	.9375	.0F	.0585 9375	.00F	.0036 6210 9375	.000F	.0002 2888 1835 9375
1		2		3		4	

TO CONVERT .ABC HEXADECIMAL TO DECIMAL

Find .A in position 1 .6250
Find .0B in position 2 .0429 6875
Find .00C in position 3 .0029 2968 7500
.ABC Hex is equal to .6708 9843 7500

Table F-5. Hexadecimal and Decimal Integer Conversion

8		7		6		5		4		3		2		1	
HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268 435 456	1	16 777 216	1	1 048 576	1	65 536	1	4 096	1	256	1	16	1	1
2	536 870 912	2	33 554 432	2	2 097 152	2	131 072	2	8 192	2	512	2	32	2	2
3	805 306 368	3	50 331 648	3	3 145 728	3	196 608	3	12 288	3	768	3	48	3	3
4	1 073 741 824	4	67 108 864	4	4 194 304	4	262 144	4	16 384	4	1 024	4	64	4	4
5	1 342 177 280	5	83 886 080	5	5 242 880	5	327 680	5	20 480	5	1 280	5	80	5	5
6	1 610 612 736	6	100 663 296	6	6 291 456	6	393 216	6	24 576	6	1 536	6	96	6	6
7	1 879 048 192	7	117 440 512	7	7 340 032	7	458 752	7	28 672	7	1 792	7	112	7	7
8	2 147 483 648	8	134 217 728	8	8 388 608	8	524 288	8	32 768	8	2 048	8	128	8	8
9	2 415 919 104	9	150 994 944	9	9 437 184	9	589 824	9	36 864	9	2 304	9	144	9	9
A	2 684 354 560	A	167 772 160	A	10 485 760	A	655 360	A	40 960	A	2 560	A	160	A	10
B	2 952 790 016	B	184 549 376	B	11 534 336	B	720 896	B	45 056	B	2 816	B	176	B	11
C	3 221 225 472	C	201 326 592	C	12 582 912	C	786 432	C	49 152	C	3 072	C	192	C	12
D	3 489 660 928	D	218 103 808	D	13 631 488	D	851 968	D	53 248	D	3 328	D	208	D	13
E	3 758 096 384	E	234 881 024	E	14 680 064	E	917 504	E	57 344	E	3 584	E	224	E	14
F	4 026 531 840	F	251 658 240	F	15 728 640	F	983 040	F	61 440	F	3 840	F	240	F	15
8		7		6		5		4		3		2		1	

TO CONVERT HEXADECIMAL TO DECIMAL

1. Locate the column of decimal numbers corresponding to the left-most digit or letter of the hexadecimal; select from this column and record the number that corresponds to the position of the hexadecimal digit or letter.
2. Repeat step 1 for the next (second from the left) position.
3. Repeat step 1 for the units (third from the left) position.
4. Add the numbers selected from the table to form the decimal number.

TO CONVERT DECIMAL TO HEXADECIMAL

1. (a) Select from the table the highest decimal number that is equal to or less than the number to be converted.
   (b) Record the hexadecimal of the column containing the selected number.
   (c) Subtract the selected decimal from the number to be converted.
2. Using the remainder from step 1(c) repeat all of step 1 to develop the second position of the hexadecimal (and a remainder).
3. Using the remainder from step 2 repeat all of step 1 to develop the units position of the hexadecimal.
4. Combine terms to form the hexadecimal number.

To convert integer numbers greater than the capacity of table, use the techniques below:

HEXADECIMAL TO DECIMAL

Successive cumulative multiplication from left to right, adding units position.

Example: $D34_{16} = 3380_{10}$

```
D =   13
    x16
    208
3 =  +3
    211
    x16
   3376
4 =  +4
   3380
```

EXAMPLE	
Conversion of Hexadecimal Value	D34
D	3328
3	48
4	4
Decimal	3380

DECIMAL TO HEXADECIMAL

Divide and collect the remainder in reverse order.

Example: $3380_{10} = D34_{16}$

```
16 | 3380 ---- 4   remainder
16 |  211 ---- 3        ^
16 |   13 ---- D        |
```

EXAMPLE	
Conversion of Decimal Value	3380
D	−3328
	52
3	−48
	4
4	−4
Hexadecimal	D34

DOCUMENT REVIEW FORM

Your comments concerning this document help us produce better documentation for you.

GENERAL COMMENTS

	Yes	No		Yes	No
Easy to Read?	☐	☐	Complete?	☐	☐
Well Organized?	☐	☐	Well Illustrated?	☐	☐
Accurate?	☐	☐	Suitable for Your Needs?	☐	☐

How do You Use this Document?

- ☐ As an introduction to the subject
- ☐ For additional knowledge
- ☐ For continual reference
- ☐ Other

SPECIFIC CLARIFICATIONS AND/OR CORRECTIONS

Reference	Page No.
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

This form should not be used as an order blank. Requests for copies of publications should be directed to the National Semiconductor sales office serving your locality.

Send comments to:

National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara, California 95051
Attention: Microprocessor Publications

National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara, California 95051
(408) 737-5000
TWX: 910-339-9240

National Semiconductor (UK) Ltd.
Larkfield Industrial Estate
Greenock, Scotland
Telephone: GOUROCK 33251
Telex: 778 632

National Semiconductor GmbH
808 Fuerstenfeldbruck
Industriestrasse 10
West Germany
Telephone: (08141) 1371
Telex: 05-27649

NS Electronics (PTE) Ltd.
No. 1100 Lower Delta Rd.
Singapore 3
Telephone: 630011
Telex: NATSEMI RS 21402

NS Electronics SDN BHD
Batu Berendam
Free Trade Zone
Malacca, Malaysia
Telephone: 5171
Telex: NSELECT 519 MALACCA
(c/o Kuala Lumpur)

PUBLICATION NO. 4200094C, ORDER NO. ISP-8S/994Y

PRINTED IN U.S.A. DD10M17